

· 开发宝典丛书 ·

一本百科全书式的C#宝典秘笈，全面、新颖、详细、深入和实用
资深程序员10年开发经验的总结，完美展现C#应用开发的精髓

C#

编程实战宝典

(15.5小时配套教学视频 + 11.5小时进阶教学视频)

付强 丁宁 等编著

- ☑ 全面：全面涵盖C#的语法、面向对象编程、Window编程、高级技术等内容
- ☑ 新颖：以当前最为流行的Visual Studio 2010作为开发平台进行讲解
- ☑ 详细：结合图示，从概念、语法、示例、技巧和应用等多角度分析每个知识点
- ☑ 实用：提供了近400个实例、40余个小案例、2个大型案例、103个习题
- ☑ 深入：深入剖析了多线程编程、XML编程、WPF、WCF、WF和LINQ等技术
- ☑ 高效：提供了15.5小时高清配套教学视频及11.5小时进阶教学视频，高效而直观

超值、大容量DVD光盘

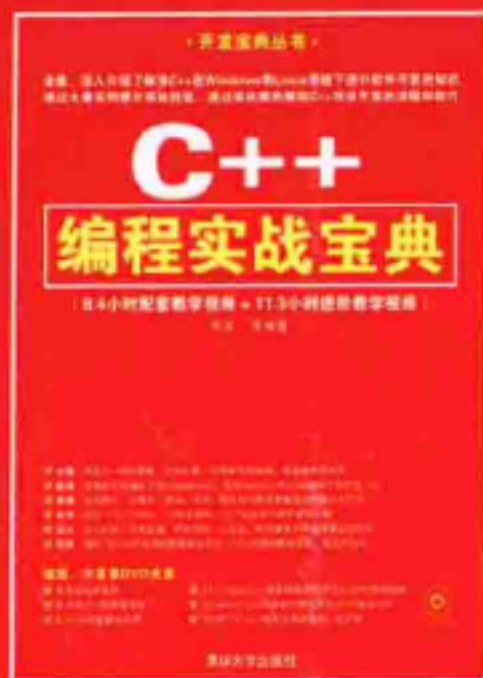
- | | |
|-------------------|---------------------------|
| ☑ 本书实例源文件 | ☑ 13个C#典型模块源程序及9小时教学视频 |
| ☑ 15.5小时高清配套教学视频 | ☑ 3个C#项目案例源程序及2.5小时教学视频 |
| ☑ 207页本书第8篇内容的电子书 | ☑ 360页《C#与.NET程序员面试宝典》电子书 |



清华大学出版社



· 开发宝典丛书 ·



下载提示

本书提供配套教学PPT，需要的读者请直接到清华大学出版社的网站上（<http://www.tup.com.cn>）搜索到本书页面后按照提示下载，也可以到本书的技术论坛上（<http://www.wanjuanchna.net>）下载。

即将上市

上架建议：计算机语言/C#

清华大学出版社数字出版网站
WQBook 书局泉
www.wqbook.com



DVD-ROM



附DVD光盘，含27小时教学视频、源文件与《C#与.NET程序员面试宝典》电子书

· 开发宝典丛书 ·

C#

编程实战宝典

付强 丁宁 等编著

清华大学出版社

内 容 简 介

本书全面、系统地介绍了使用 C# 语言进行开发的方方面面知识。书中的各个技术点都提供了实例供读者实践练习，各章后还提供了实战练习题帮助读者巩固和提高。本书中的每个例子都经过精挑细选，具有较强的针对性，力求使读者通过书中的示例能够更迅速地掌握相关知识。本书配 1 张 DVD 光盘，内容为 15.5 小时高清配套教学视频及本书涉及的实例源文件，光盘中还赠送了一部.NET 面试宝典电子书和大量的 C# 开发范例、模块和项目案例的源程序及教学视频等资料。

本书共 32 章，分为 8 篇。第 1 篇介绍了 Visual Studio 2010 的开发环境、发展历史和应用范围等；第 2 篇介绍了 C# 语言的基础知识，如数据类型、变量与表达式、程序控制语言、函数与方法等；第 3 篇介绍了面向对象的基础知识及其在 C# 中的新应用和特征；第 4 篇介绍了 Windows 应用程序设计，对 Visual Studio 所提供的控件进行了较为全面的介绍；第 5 篇介绍了 C# 的高级特性和工具，如异常处理、文件系统与流、XML 和多线程编程等；第 6 篇主要介绍了 Web 数据库开发的相关知识；第 7 篇介绍了 WCF、WPF、WF 和 LINQ 这 4 个 .NET 4.0 中新增的功能；第 8 篇介绍了音乐商店网站和电子购物商城两个项目案例的总体设计和实现（因篇幅所限，本篇内容以 PDF 电子文档的格式收录于本书的配书光盘中）。

本书适合想全面学习 C# 编程和使用 C# 进行开发的工程技术人员阅读。对于 .NET 程序员，本书更是一本不可多得的案头必备参考手册。另外，本书可作为计算机和软件工程等专业的教材和教学参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目（CIP）数据

C# 编程实战宝典 / 付强等编著. — 北京：清华大学出版社，2014

（开发宝典丛书）

ISBN 978-7-302-35517-5

I. ①C… II. ①付… III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2014）第 032462 号

责任编辑：夏兆彦

封面设计：欧振旭

责任校对：徐俊伟

责任印制：何 芊

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社总机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者：清华大学印刷厂

经 销：全国新华书店

开 本：185mm×260mm 印 张：54.75 字 数：1370 千字

（附光盘 1 张）

版 次：2014 年 9 月第 1 版

印 次：2014 年 9 月第 1 次印刷

印 数：1~3000

定 价：99.80 元

产品编号：056425-01

前言

Visual Studio 是一套完整的开发工具，它可以用于开发桌面应用程序、ASP Web 应用程序、XML Web 服务及移动应用程序等。而 C#（读 C sharp）是微软开发的一种专门运行于 Visual Studio 上的开发语言。它是从 C 语言和 C++ 语言派生的一种简单且面向对象和类型安全的编程语言。在 .NET Framework 环境下，C# 结合了 Visual Basic 的快速开发能力和 C++ 强大而灵活的能力，使得程序员可以快速编写出各种基于 Microsoft .NET 平台的应用程序。而 C# 面向对象的良好特征，也使得它成为了构建各种应用程序的最佳选择。

为了帮助众多想全面学习 C# 语言的读者能够快速、准确地掌握该编程语言，笔者精心编写了本书。本书在内容编排上遵循科学的学习规律，争取让读者能够快速掌握这门语言。例如，书中开始先熟悉了基本的开发界面，然后便以一个实例演示如何使用 C# 语言编写出一个简单的 Windows 应用程序，让读者快速入门，也顺便了解一下 Windows 应用程序开发的基本流程。在讲解具体知识的时候，本书尽量避免冗长的理论讲解，而是通过具体实例直接告诉读者本章所讲的内容是如何在应用程序中得以实现的，便于读者在实际操作中学习 C# 语言。这种边学边练的方式，让读者不仅加深了对理论知识的理解，更为重要的是提高了动手编程能力。另外，笔者还专门为本书内容录制了大量高清配套教学视频以辅助读者学习，从而达到更好的学习效果。

相信通过本书，读者能够较为全面地掌握 C# 程序设计的各种语法功能和技巧，为后续进一步学习 .NET 程序开发打好基础。而且，相信通过本书，读者可以在以后的 IT 求职面试和程序开发工作中都有章可循，游刃有余。

本书特色

1. 实例具体，内容充实

本书不像其他教程那样只单纯讲解 C# 语言的语法规则，而是通过与笔者在开发和面试过程中经常遇到的需求相结合，将 C# 语言的基本功能和操作技巧融入到示例当中进行讲解，并全面覆盖语法知识、面向对象、Windows 编程、高级编程等 C# 语言的重点内容。

2. 基于需求，面向求职

作者在讲解每一个知识点之前，充分考虑了将 C# 语言的知识与实践工作相结合，精心挑选出了符合各类常见程序需求的开发实例，使读者不仅能学到 C# 语言的知识，而且可以了解实际的 IT 求职面试和工作中的要求。

3. 图文并茂，步骤详细

书中每个实例的实现步骤都以通俗易懂的语言阐述，并配有插图，详细而贴切。读者只需要按照步骤操作，就可以学习到 C#语言的相关知识，并体会到独立开发的乐趣。

4. 案例精讲，注重实战

本书第 8 篇详细介绍了两个综合项目案例的开发过程，以提高读者的实际开发水平，从而轻松地应对实际的项目开发。

5. 实践练习，巩固提高

本书各章最后都提供了典型的实践练习题，读者每阅读完一章，可以通过完成这些练习题来检测自己的学习效果，从而达到巩固和提高的效果。

6. 视频教学，加速学习

为了让读者的学习更加直观和高效，作者为本书录制了 15.5 小时多媒体教学视频。读者在阅读本书时可以结合光盘中的教学视频，从而达到更好的学习效果。

本书内容

第 1 篇 开发环境及 C#语言简介（第 1、2 章）

本篇首先向读者介绍了 Visual Studio 2010 的开发环境、发展历史、适用范围，然后简述了 C#语言的基本特点及与其他语言相比较的优点，最后通过举例介绍了使用 C#语言进行 Windows 应用程序开发的一般流程。

第 2 篇 C#程序设计基础（第 3~6 章）

本篇切入正题，介绍了 C#语言的基础知识。首先介绍了 C#语言的基础数据、基本操作符、数据间的转换和常量与变量，然后讲述了 C#中的程序控制语句，最后介绍了 C#中的函数与方法。

第 3 篇 C#面向对象编程简介（第 7~11 章）

本篇 C#对面向对象编程的相关知识进行了详细的介绍。内容主要包括对象和类、继承与多态技术、抽象类和接口及 C#特有的代理和时间。另外，本篇还介绍了 C#中的数组和集合。

第 4 篇 Windows 程序设计（第 12~18 章）

本篇是 Windows 应用程序设计部分，这部分内容在 C#开发中占有非常重要的地位。本篇通过逐一介绍 Visual Studio 所提供的常用控件，来讲解程序开发中经常会遇到的应用需求。可以说，本书前 3 篇是本篇的基础，而本篇是前 3 篇的综合。

第 5 篇 C#高级编程技术和工具（第 19~22 章）

本篇介绍了 C#高级编程的相关技术。主要介绍了 C#语言的异常处理、流与文件系统、可扩展标记语言及多线程编程等。随着读者编程水平的提高，本篇内容的重要性也就逐渐体现了出来。请读者在阅读时勤加思考，与书中的具体示例结合起来学习。

第 6 篇 Web 数据库开发（第 23~26 章）

本篇主要介绍了数据库基础知识、ADO.NET 数据库编程、ASP.NET 技术入门和服务端控件等内容。

第 7 篇 .NET 4.0 增强功能（第 27~30 章）

本篇主要介绍了.NET 4.0 的增强功能，包括 WPF、WCF、WF 和 LINQ 等技术。这些都是.NET 技术的最新亮点，体现了.NET 技术的发展趋势。

第 8 篇 综合项目案例实战（第 31、32 章）

本篇着重讲解了两个实际的综合项目案例的开发过程，一个是用 ASP.NET MVC 开发的音乐商店网站，另一个是开发电子商务网站。因篇幅所限，本篇内容以 PDF 电子文档的格式收录于本书的配书光盘中。

本书超值 DVD 光盘内容

- ☐ 本书各章涉及的实例源文件；
- ☐ 15.5 小时本书配套教学视频；
- ☐ 13 个 C#典型模块源程序及 9 小时教学视频；
- ☐ 3 个 C#项目案例源程序及 2.6 小时教学视频；
- ☐ 207 页本书第 8 篇内容的电子书；
- ☐ 360 页《C#与.NET 程序员面试宝典》电子书。

本书读者对象

- ☐ C#编程入门人员；
- ☐ 想全面学习 C#编程的人员；
- ☐ C#编程爱好者；
- ☐ C#专业开发人员；
- ☐ 利用 C#做开发的工程技术人员；
- ☐ 大中专院校的学生；
- ☐ 社会培训班的学员；
- ☐ 需要作为案头必备手册的程序员。

本书光盘内容

- 本书配套多媒体教学视频;
- 本书涉及的实例源文件;
- 本书第 31 章和 32 章的 PDF 文档;
- 《C#与.NET 程序员面试宝典》电子书;
- C#范例开发视频与源码库;
- C#典型模块开发视频与源码库;
- C#项目案例开发视频与源码库。

本书作者

本书由付强、丁宁主笔编写。其他参与编写的人员有陈鹏、段丽钢、侯晶晶、侯亮亮、侯荣、侯瑞雪、胡慧、李慧敏、卢剑辉、沈炜、孙景瑞、王大中、吴艳敏、闫刚、张琼、张笑、朱万江、刘素兵、张华、王冬姣、刘应舫、关黎霞、曾源、池剑锋、郝军、周凌霄、葛亮、郭颂、金楠、刘岩、马军、王晋凯、武松、张国平、张敬普、伍远高、周颖。


本书的编写对我们而言是一个“浩大的工程”。虽然我们投入了大量的精力和时间，但只怕百密难免一疏，书中可能还存在一些疏漏。若读者在阅读本书时发现任何疏漏，希望能及时反馈给我们，以便及时更正。联系我们请发邮件至 bookservice2008@163.com。


最后祝各位读者读书快乐，学习进步！

编著者

目 录


第 1 篇 开发环境及 C#语言简介

第 1 章 Visual Studio 2010 介绍 ( 教学视频: 31 分钟)	2
1.1 .NET 概述	2
1.1.1 .NET 的前世今生	2
1.1.2 什么是微软中间语言	3
1.1.3 背后默默付出的垃圾收集器	4
1.2 .NET Framework 概述	5
1.2.1 .NET Framework 包含什么	5
1.2.2 .NET 平台的核心: CLR	7
1.2.3 基类库是神马	7
1.2.4 公共语言运行规范	9
1.3 Visual Studio 2010 有哪些新增功能	9
1.4 Visual Studio 2010 开发环境介绍	11
1.4.1 安装 Visual Studio 2010	11
1.4.2 Visual Studio 2010 提供哪些项目模板	13
1.4.3 Visual Studio 2010 提供哪些网站模板	15
1.4.4 Visual Studio 2010 提供哪些文件模板	15
1.4.5 与 Visual Studio 2010 的第一次相会	16
1.4.6 必须熟悉的开发工具栏和菜单栏	17
1.4.7 用对象浏览器查看对象信息	20
1.4.8 可视化利器: 工具箱和属性窗口	21
1.5 定制环境	23
1.5.1 让字体和颜色更适合自己	23
1.5.2 在项目和解决方案中保存文件	23
1.5.3 使用任务列表和注释管理代码	24
1.5.4 在命令窗口中执行命令	24
1.5.5 代码显示行号与代码折叠	26
1.5.6 管理 Visual Studio 2010 中的子窗口	27
1.5.7 调试与生成程序	28
1.6 学会使用 MSDN 帮助系统	30

1.7	用 Visual Studio 2010 编写第一个程序	31
1.8	本章总结	33
1.9	实战练习	34
第 2 章	C#简介 ( 教学视频: 16 分钟)	35
2.1	C#与.NET 的关系	35
2.2	C#有哪些特点	35
2.2.1	简单性	36
2.2.2	类型统一性	38
2.2.3	面向对象性	39
2.2.4	类型安全性	40
2.2.5	兼容性	41
2.3	C#与其他语言对比	41
2.4	C#与 VB.NET 的异同	42
2.4.1	代码表现形式的差异	42
2.4.2	数据类型和变量使用的差异	43
2.4.3	类、数据类型、函数以及接口	44
2.4.4	操作符与表达式的差异	44
2.4.5	控制流程语句的差异	45
2.4.6	错误处理的差异	47
2.4.7	关键字的差异	47
2.4.8	访问修饰符的差异	47
2.4.9	语法的差异	48
2.4.10	C#与 VB.NET 实例对比	48
2.5	C#与 C++的异同	51
2.5.1	托管环境的差异	51
2.5.2	C#使用.NET 的对象	52
2.5.3	C#的语句	52
2.5.4	C#中取消的要素	52
2.5.5	操作符重载的差异	52
2.5.6	头文件的差异	53
2.5.7	程序书写的差异	53
2.5.8	被取消的指针	53
2.5.9	虚函数的差异	53
2.5.10	C#与 C++实例对比	55
2.6	C#与 Java 的异同	56
2.6.1	数据类型的差异	56
2.6.2	类的差异	57
2.6.3	属性定义的差异	58
2.6.4	事件、指针与界面的差异	59
2.6.5	C#与 Java 实例对比	59

2.7 本章总结	60
2.8 实战练习	61

第 2 篇 C#程序设计基础

第 3 章 C#数据类型 ( 教学视频: 37 分钟)	64
3.1 初识 C#的数据类型	64
3.2 存储实际数据的值类型	65
3.2.1 什么是值类型	65
3.2.2 整型	66
3.2.3 字符型	67
3.2.4 浮点型	68
3.2.5 小数型	69
3.2.6 大整数型	69
3.2.7 复数型	70
3.2.8 布尔型	71
3.2.9 C#值类型的数值类型	71
3.3 存储引用地址的引用类型	72
3.3.1 什么是引用类型	72
3.3.2 类类型	72
3.3.3 对象类型	73
3.3.4 字符串类型	73
3.3.5 接口类型	75
3.3.6 dynamic 类型	76
3.4 数据类型是可以转换的	78
3.4.1 什么时候发生类型转换	78
3.4.2 显式转换	78
3.4.3 隐式转换	79
3.4.4 不同数值类型之间的转换	79
3.4.5 数值类型和字符串之间的转换	80
3.4.6 字符的 ASCII 码和 Unicode 码之间的转换	82
3.4.7 字符串和字符数组之间的转换	84
3.4.8 字符串和字节数组之间的转换	85
3.4.9 数值类型和字节数组之间的转换	86
3.4.10 不同类型之间的强制转换	87
3.5 C#的用户自定义数据类型	88
3.5.1 认识枚举类型	88
3.5.2 枚举编程示例	89
3.5.3 认识结构类型	90

3.5.4	结构编程示例	90
3.5.5	结构也支持方法	92
3.5.6	结构与类有什么不同	93
3.5.7	哪些地方应使用结构类型	94
3.6	本章总结	94
3.7	实战练习	94
第 4 章	变量与表达式 (📺 教学视频: 42 分钟)	96
4.1	常量与变量	96
4.1.1	什么是常量	96
4.1.2	静态常量的特点	96
4.1.3	动态常量的特点	97
4.1.4	该用静态常量还是动态常量	97
4.1.5	什么是变量	98
4.1.6	认识变量的实质	99
4.1.7	变量有哪些种类	100
4.1.8	怎样给变量命名	102
4.2	连接的桥梁——运算符与表达式	102
4.2.1	C#的运算符分类	102
4.2.2	算术运算符	103
4.2.3	关系运算符	105
4.2.4	一般赋值运算符	106
4.2.5	复合赋值运算符	107
4.2.6	逻辑运算符	109
4.2.7	复习二进制知识	112
4.2.8	二进制的位运算符	113
4.2.9	有逻辑判断功能的三元运算符	115
4.2.10	自增和自减运算符	116
4.2.11	应该先进行什么运算	117
4.3	关键字	118
4.4	本章总结	119
4.5	实战练习	119
第 5 章	程序控制语言 (📺 教学视频: 46 分钟)	120
5.1	C#有哪些种类的语句	120
5.2	选择语句让程序具有智能	121
5.2.1	选择语句的作用	121
5.2.2	认识 if 语句	121
5.2.3	单分支 if 语句	122
5.2.4	二分支 if 语句	123
5.2.5	多分支 if 语句	124

5.2.6	if 语句多层嵌套	125
5.2.7	switch 多分支选择语句	127
5.2.8	switch 语句编程示例	128
5.3	用循环语句进行重复劳动	129
5.3.1	do 循环语句	129
5.3.2	while 循环语句	130
5.3.3	for 循环语句	131
5.3.4	foreach 循环语句	133
5.4	用跳转语句改变程序流程	134
5.4.1	用 break 语句跳出循环	134
5.4.2	用 continue 语句进入下次循环	135
5.4.3	用 return 语句返回	136
5.4.4	用 goto 语句跳到指定行	137
5.4.5	用 goto 语句跳出 switch 语句	138
5.4.6	用 goto 语句跳出一层嵌套循环	139
5.4.7	用 throw 语句抛出异常	141
5.5	用注释语句让代码意图更明了	142
5.5.1	普通注释语句	142
5.5.2	可生成帮助文档的注释语句	143
5.6	本章总结	146
5.7	实战练习	146
第 6 章	函数与方法 ( 教学视频: 38 分钟)	147
6.1	函数是 C# 的基本结构	147
6.1.1	函数与方法	147
6.1.2	无参和有参函数	147
6.2	函数的必备件: 参数与返回值	148
6.2.1	参数有什么用	148
6.2.2	值参数的使用	148
6.2.3	引用参数的使用	149
6.2.4	输出参数的使用	152
6.2.5	数组参数的使用	153
6.2.6	命名参数和可选参数	154
6.2.7	利用返回值获取数值	156
6.2.8	利用返回值判断逻辑	157
6.3	变量的作用域	159
6.3.1	最常见的局部变量	159
6.3.2	需要慎用的全局变量	160
6.4	认识主入口函数 Main()	161
6.5	C# 中最常用的函数	165
6.6	本章总结	166

6.7 实战练习	166
----------	-----

第3篇 C#面向对象编程简介

第7章 类和对象 (🔑 教学视频: 33 分钟)	170
7.1 类和对象的关系	170
7.2 类的定义	171
7.2.1 创建一个类	171
7.2.2 类成员的访问类型	173
7.2.3 创建类的成员方法	174
7.3 创建对象	175
7.3.1 类的构造函数有什么用	175
7.3.2 成员变量的初始化	177
7.3.3 创建类的拷贝构造函数	179
7.3.4 用关键字 this 引用当前对象	179
7.4 静态成员的使用	180
7.4.1 什么是类的静态成员	180
7.4.2 静态方法怎样调用	181
7.4.3 静态构造函数的优势	182
7.4.4 使用私有构造函数保护静态成员	182
7.4.5 使用静态成员变量记录对象数量	183
7.5 对象的销毁	184
7.5.1 C#的析构函数	184
7.5.2 用 Dispose() 方法释放资源	185
7.5.3 用 using 声明作用范围	185
7.6 参数传递	186
7.6.1 通过引用传递返回多个值	186
7.6.2 用 out 类型参数返回值	189
7.7 重写方法和构造函数让类更宜用	191
7.8 用属性封装类的数据	193
7.8.1 类的属性定义	193
7.8.2 用 get 访问器返回属性值	195
7.8.3 用 set 访问器设置属性值	196
7.9 用 Readonly 类型变量为类设置不变的值	196
7.10 本章总结	198
7.11 实战练习	199
第8章 继承与多态 (🔑 教学视频: 39 分钟)	200
8.1 用继承对类进行扩展	200


8.1.1	面向对象程序中的继承	200
8.1.2	C#的继承机制	201
8.1.3	继承是怎样调用构造函数	205
8.1.4	用 base 关键字调用基类构造函数	206
8.1.5	使用 protected 访问类型	208
8.1.6	用 sealed 关键字终止继承	209
8.2	万类之根——Object 类	210
8.3	C#类的多态特性	212
8.3.1	什么是多态	212
8.3.2	在基类与子类中定义同名方法	213
8.3.3	方法重写——virtual 和 override 关键字的使用	214
8.3.4	用 new 关键字指出重写方法	216
8.4	对象类型转换——Up-casting 和 Down-casting	217
8.4.1	子类到基数的 Up-casting 类型转换	217
8.4.2	基数到子类的 Down-casting 类型转换	217
8.4.3	用 is 和 as 关键字做对象类型检查	218
8.5	装箱和拆箱	219
8.6	本章总结	219
8.7	实战练习	220
第 9 章	抽象类和接口 (🎥 教学视频: 24 分钟)	221
9.1	定义抽象类	221
9.2	定义统一规划的接口	224
9.2.1	定义接口	224
9.2.2	一个类实现多个接口	227
9.2.3	实现多接口同名方法	229
9.2.4	使用 is 和 as 操作符实现接口转换	231
9.2.5	接口间能继承吗	232
9.3	本章总结	234
9.4	实战练习	234
第 10 章	数组与集合 (🎥 教学视频: 38 分钟)	235
10.1	数组的数组——多维数组	235
10.1.1	认识多维数组	235
10.1.2	怎样实例化多维数组	236
10.1.3	什么是变长数组	238
10.2	用 foreach 循环进行数组的遍历	240
10.3	比数组灵活好用的集合	243
10.3.1	用 ArrayList 集合保存不同类型数据	244
10.3.2	用 Stack 集合处理栈	246
10.3.3	用 Queue 集合处理队列	248


10.4 键-值对应形式的字典类型集合	249
10.4.1 Hashtable 类存储方式	250
10.4.2 创建 Hashtable 集合的方法	250
10.4.3 用 Add 方法向 Hashtable 集合中添加元素	250
10.4.4 从 Hashtable 集合中获取元素的方法	251
10.4.5 用 Remove 方法移除 Hashtable 集合中的元素	251
10.4.6 获取 Hashtable 集合的元素与键值	252
10.4.7 在 Hashtable 集合中检索元素	253
10.4.8 SortedList 类与 Hashtable 类的区别	253
10.4.9 SortedList 类常用方法	253
10.5 本章总结	255
10.6 实战练习	256
第 11 章 代理和事件 (🔗 教学视频: 31 分钟)	257
11.1 代理	257
11.1.1 什么是代理	257
11.1.2 代理所指向方法的类型和标识	258
11.1.3 代理引用的声明和使用	258
11.1.4 .NET Framework 中的代理	262
11.1.5 代理用作方法的参数	262
11.1.6 了解多重代理	264
11.1.7 多重代理的实现方法	264
11.1.8 怎样移除多重代理指向的方法	266
11.2 事件和事件处理	267
11.2.1 C#的事件处理	267
11.2.2 事件举例——时钟事件	268
11.2.3 多重事件的处理	270
11.2.4 利用事件传递数据	271
11.3 本章总结	274
11.4 实战练习	275


第 4 篇 Windows 程序设计

第 12 章 Windows 应用程序概述 (🔗 教学视频: 29 分钟)	278
12.1 Windows 应用程序	278
12.1.1 什么是 Windows 应用程序	278
12.1.2 创建 Windows 应用程序	278
12.1.3 程序入口文件 Program.cs	279
12.1.4 窗口程序文件 Form1.cs	279


12.1.5	窗口设计文件 Form1.Designer.cs	280
12.2	Windows 应用程序窗体	281
12.2.1	C#的 Form 类	281
12.2.2	Form 类常用属性	281
12.2.3	Form 类常用事件	282
12.3	为窗体添加控件	283
12.3.1	Windows 窗体控件	283
12.3.2	控件常见属性	284
12.3.3	控件常用公共事件	284
12.3.4	向窗体添加控件的方法	285
12.3.5	调整控件的布局	285
12.4	Windows 应用程序编程示例	287
12.4.1	创建一个窗体	287
12.4.2	用属性控制窗体外观	287
12.4.3	向窗体添加控件	288
12.4.4	添加控件事件处理程序	288
12.4.5	查看窗体运行效果	288
12.5	本章总结	289
12.6	实战练习	289
第 13 章	Visual Studio 2010 控件介绍 ( 教学视频: 39 分钟)	290
13.1	接收输入的文本编辑控件	290
13.1.1	TextBox 控件的作用	290
13.1.2	怎样获取或设置 TextBox 控件的内容	291
13.1.3	TextBox 控件也可输入多行文本	292
13.1.4	选择 TextBox 控件中的文本	293
13.1.5	设置 Textbox 控件为密码框	294
13.1.6	检查 TextBox 控件的输入值	295
13.1.7	支持格式化的 RichTextBox 控件	295
13.1.8	选择 RichTextBox 控件中的文本	296
13.1.9	在 RichTextBox 控件中撤销上次操作	297
13.1.10	拖放 RichTextBox 控件中的文本	298
13.1.11	设置 RichTextBox 控件中的文本格式	299
13.1.12	设置 RichTextBox 控件的文本缩进	300
13.1.13	在 RichTextBox 控件中添加超链接	302
13.1.14	RichTextBox 控件的文件操作	302
13.2	不能编辑的文本显示控件	304
13.2.1	用 Label 控件显示文本	304
13.2.2	用 LinkLabel 控件显示超链接文本	305
13.2.3	保存超链接的 LinkCollection 集合	307
13.2.4	LinkLabel 控件编程示例	307

13.3	包容其他控件的容器控件	309
13.3.1	用 Panel 控件细分窗体	309
13.3.2	有标题的容器控件 GroupBox	312
13.3.3	多页容器控件 TabControl	313
13.3.4	TabControl 控件的几种常见外观	314
13.3.5	TabControl 控件中标签页的选择	316
13.3.6	添加 TabControl 控件中的标签页	318
13.3.7	网格形式容器控件 TableLayoutPanel	320
13.3.8	设置 TableLayoutPanel 控件的外观	321
13.3.9	添加控件到 TableLayoutPanel 控件	323
13.4	值设置控件	325
13.4.1	用 CheckBox 控件选择打开或关闭状态	325
13.4.2	用 RadioButton 控件进行多选一操作	328
13.5	本章总结	329
13.6	实战练习	329
第 14 章	列表选择控件介绍 ( 教学视频: 43 分钟)	330
14.1	ListBox 控件	330
14.1.1	用 ListBox 显示列表	330
14.1.2	用 ListBox 添加列表项	330
14.1.3	设置 ListBox 的行为	332
14.1.4	让 ListBox 支持多选	334
14.1.5	设置 ListBox 的外观	336
14.1.6	对 ListBox 列表进行排序	337
14.1.7	控制 ListBox 列表项的刷新	337
14.1.8	查找 ListBox 中的列表	338
14.1.9	Listbox 控件的常用方法	340
14.2	ComboBox 控件	342
14.2.1	认识 ComboBox 控件	342
14.2.2	设置 ComboBox 下拉样式	342
14.2.3	设置 ComboBox 的自动补齐	344
14.2.4	ComboBox 自动补齐的数据源	344
14.2.5	设置 ComboBox 自动补齐方式	347
14.2.6	ComboBox 的常见事件	348
14.2.7	修改 ComboBox 的子项	349
14.2.8	ComboBox 的子项绘制	350
14.3	CheckedListBox 控件	353
14.3.1	带复选标记的列表控件 CheckedListBox	353
14.3.2	CheckedListBox 控件编程示例	355
14.4	ListView 控件	359
14.4.1	带图标的列表控件 ListView	359

14.4.2	ListView 的 5 种视图	360
14.4.3	使用任务窗体设置 ListView	361
14.4.4	用 ImageList 给 ListView 提供图标	362
14.4.5	设置 ListView 的子项	364
14.4.6	设置 Details 视图模式的数据	365
14.4.7	给 ListView 添加分组设置	367
14.4.8	让 ListView 支持拖曳操作	368
14.4.9	在 ListView 中进行搜索	370
14.5	TreeView 控件	371
14.5.1	用 TreeView 控件显示分层数据	371
14.5.2	添加节点到 TreeView	372
14.5.3	在节点前显示复选框	374
14.5.4	选中节点的常用约定	376
14.5.5	用 ImageList 为 TreeView 提供图标	376
14.5.6	展开与折叠 TreeView 的节点	377
14.5.7	允许编辑 TreeView 的节点文本	379
14.5.8	让 TreeView 支持拖曳操作	380
14.6	本章总结	383
14.7	实战练习	383
第 15 章	数据显示控件 ( 教学视频: 27 分钟)	384
15.1	DataGridView 控件以表格显示数据	384
15.1.1	DataGridView 有哪些功能	384
15.1.2	设置对表格数据的操作	385
15.1.3	在窗体中如何放置 DataGridView	388
15.1.4	为 DataGridView 提供数据	389
15.1.5	向 DataGridView 添加列	389
15.1.6	冻结 DataGridView 中的列	390
15.1.7	编辑 DataGridView 中的列	391
15.2	为 DataGridView 服务的类	392
15.2.1	DataGridViewElement 对象模型	392
15.2.2	单元格对象模型 DataGridViewCell	393
15.2.3	数据列对象模型 DataGridViewColumn	394
15.2.4	数据行对象模型 DataGridViewRow	394
15.3	设置 DataGridView 的格式	395
15.3.1	设置单元格边框样式	395
15.3.2	设置 DataGridView 控件边框样式	396
15.3.3	设置单元格边框样式	397
15.3.4	设置左上角单元格边框样式	398
15.3.5	设置单元格样式	399
15.3.6	单元格样式设置的优先级	402

15.3.7	设置 DataGridView 外观	404
15.3.8	用属性设置可对表格的操作	404
15.3.9	DataGridView 其他常用属性	405
15.3.10	用 AutoResizeColumn 方法调列宽	407
15.3.11	用 AutoResizeColumns 方法调整所有列宽	409
15.3.12	用 AutoResizeColumnHeadersHeight 方法调整列标题高度	410
15.3.13	用 AutoResizeRowHeadersWidth 方法调整行标题宽度	411
15.3.14	DataGridView 其他常用方法	412
15.4	DataGridView 中显示数据的相关类	413
15.4.1	设置数据排序模式	413
15.4.2	用 DataGridViewTextBoxColumn 显示文本数据列	413
15.4.3	用 DataGridViewCheckBoxColumn 显示二元状态数据列	415
15.4.4	用 DataGridViewImageColumn 显示图像数据列	416
15.4.5	用 DataGridViewButtonColumn 显示按钮数据列	418
15.4.6	用 DataGridViewComboBoxColumn 显示下拉列表数据列	419
15.4.7	用 DataGridViewLinkColumn 显示超链接数据列	421
15.5	本章总结	423
15.6	实战练习	423
第 16 章	通用对话框 ( 教学视频: 40 分钟)	424
16.1	消息对话框	424
16.1.1	用消息对话框显示信息	424
16.1.2	该怎样显示消息对话框	425
16.1.3	消息对话框能显示哪些按键	426
16.1.4	用户按了哪个按钮	427
16.1.5	消息对话框可显示哪些图标	429
16.1.6	消息对话框编程示例	430
16.2	文件对话框	433
16.2.1	认识 OpenFileDialog 组件	433
16.2.2	显示“打开文件”对话框	434
16.2.3	打开快捷方式引用的文件	434
16.2.4	同时选择多个文件	435
16.2.5	设置可打开的文件类型	436
16.2.6	设置“打开”文件对话框的外观	438
16.2.7	检查指定的文件是否存在	440
16.2.8	使用 SaveFileDialog 组件	441
16.2.9	SaveFileDialog 组件编程示例	442
16.2.10	另存文件时的覆盖与新建	444
16.2.11	使用 FolderBrowserDialog 组件	444
16.2.12	FolderBrowserDialog 组件编程示例	445
16.3	字体选择对话框	448

16.3.1	使用 FontDialog 组件	448
16.3.2	FontDialog 组件编程示例	449
16.4	颜色选择对话框	451
16.4.1	使用 ColorDialog 组件	451
16.4.2	ColorDialog 组件编程示例	452
16.5	打印相关对话框	454
16.5.1	使用 PrintDocument 组件	454
16.5.2	PrintDocument 组件编程示例	454
16.5.3	使用 PageSetupDialog 组件	457
16.5.4	PageSetupDialog 组件编程示例	458
16.5.5	使用 PrintPreviewDialog 组件	458
16.5.6	PrintPreviewDialog 组件编程示例	460
16.5.7	使用 PrintDialog 组件	462
16.6	本章总结	463
16.7	实战练习	464
第 17 章	其他常用控件 ( 教学视频: 29 分钟)	465
17.1	计时器组件	465
17.1.1	Timer 组件有什么用	465
17.1.2	用 Timer 控制程序定时执行	466
17.1.3	用 Timer 制作秒表	467
17.1.4	启动和重置 Timer	469
17.2	图形控件	470
17.2.1	用 PictureBox 控件显示图片	470
17.2.2	PictureBox 显示图片的 5 种方式	471
17.2.3	更新 PictureBox 中的图片	472
17.2.4	用 ImageList 存储图片	473
17.2.5	哪些控件可用 ImageList 中的图片	474
17.3	菜单控件	475
17.3.1	用 MenuStrip 创建菜单	475
17.3.2	有哪些方法添加菜单项	476
17.3.3	为菜单项编写代码完成相应功能	479
17.3.4	用 ToolStrip 创建工具栏	481
17.3.5	工具栏中可使用的按钮类型	482
17.3.6	设置工具栏的外观和行为	483
17.3.7	设置 ToolStripComboBox 的自动完成功能	485
17.3.8	设置 ToolStrip 的外观	486
17.3.9	用 ContextMenuStrip 创建快捷菜单	486
17.4	本章总结	487
17.5	实战练习	487

第 18 章 Windows 应用程序的部署 ( 教学视频: 22 分钟)	489
18.1 你了解这两种部署方案吗	489
18.1.1 什么是 ClickOnce 部署方案	489
18.1.2 什么是 Windows Installer 部署方案	490
18.2 ClickOnce 部署	490
18.2.1 创建一个部署用的示例程序	490
18.2.2 启动“发布向导”的 3 种方法	491
18.2.3 设置发布应用程序的 3 个 URL 地址	492
18.2.4 为应用程序设置名称	494
18.2.5 设置需发布的非代码文件	494
18.2.6 为应用程序添加下载组	495
18.2.7 设置应用程序的系统必备组件	495
18.2.8 设置安装组件的权限	496
18.2.9 应用程序的 3 种发布方式	497
18.2.10 将应用程序发布到网站	497
18.2.11 将应用程序发布到共享文件夹	499
18.2.12 将应用程序发布到 CD-ROM	500
18.2.13 发布应用程序	501
18.2.14 安装发布的应用程序	503
18.3 Windows Installer 部署	504
18.3.1 创建一个部署用的示例程序	504
18.3.2 创建部署项目	505
18.3.3 设置文件安装到目标机器的位置	505
18.3.4 添加项目输出组	506
18.3.5 设置安装后的桌面快捷方式	507
18.3.6 更改安装应用程序的部署特性	508
18.3.7 将应用程序信息添加到注册表	509
18.3.8 安装过程中可使用的对话框	509
18.3.9 “欢迎使用”用户界面对话框	510
18.3.10 “选择安装文件夹”用户界面对话框	511
18.3.11 “确认安装”用户界面对话框	511
18.3.12 “进度”用户界面对话框	512
18.3.13 “安装完成”用户界面对话框	512
18.3.14 安装过程中的自定义默认对话框	513
18.3.15 “单选按钮”用户界面对话框	514
18.3.16 “复选框”用户界面对话框	515
18.3.17 “文本框”用户界面对话框	516
18.3.18 “客户信息”用户界面对话框	517
18.3.19 “许可协议”用户界面对话框	517
18.3.20 “自述文件”用户界面对话框	518

18.3.21 “注册用户”用户界面对话框	518
18.3.22 生成应用程序安装文件	519
18.4 比较两种部署方案	519
18.5 本章总结	520
18.6 实战练习	520

第 5 篇 C#高级编程技术和工具


第 19 章 异常处理 (🎥 教学视频: 28 分钟)	522
19.1 程序运行中的	522
19.2 C#和.NET 中的异常处理	524
19.2.1 使用 try...catch...finally 结构处理异常	524
19.2.2 捕获程序中可能产生的异常	526
19.2.3 用 finally 语段释放资源	528
19.3 多异常的捕获	530
19.3.1 什么多异常	530
19.3.2 异常的继承关系	533
19.3.3 捕获所有异常的方法	534
19.4 定义用户异常的方法	534
19.5 本章总结	537
19.6 实战练习	537
第 20 章 文件系统与流 (🎥 教学视频: 45 分钟)	538
20.1 软件系统环境相关信息	538
20.1.1 用 System.Environment 类获得应用程序运行环境的信息	538
20.1.2 System.Environment 类的应用举例	538
20.1.3 用 Enviroment.GetFolderPath()获得各种 Windows 标准文件夹的路径	540
20.2 对文件进行操作	542
20.2.1 C#对文件进行操作的类	542
20.2.2 用 System.IO.File 类的静态方法操作文件	542
20.2.3 用 System.IO.FileInfo 类的方法操作文件	544
20.3 对文件夹进行操作	545
20.3.1 C#对文件夹操作的类	545
20.3.2 用 System.IO.Directory 类的静态方法操作文件夹	545
20.3.3 用 System.IO.DirectoryInfo 类的方法操作文件夹	547
20.4 流文件概述	549
20.4.1 了解流	549
20.4.2 Stream 类的常用方法和属性	549
20.5 使用流对文件进行读写	550


20.5.1	使用 System.IO.FileStream 类进行文件读写	550
20.5.2	用 System.IO.FileStream 类打开文本文件	551
20.5.3	用 BinaryReader 和 BinaryWriter 类读写原始文件	553
20.5.4	用 StreamReader 和 StreamWriter 类读写文本文件	554
20.6	序列化和反序列化	554
20.6.1	什么是对象序列化	555
20.6.2	用格式器描述被序列化对象	555
20.6.3	对象序列化使用示例一	555
20.6.4	对象序列化使用示例二	558
20.6.5	对象序列化使用示例三	560
20.7	异步读取数据	562
20.7.1	什么是异步读取数据	562
20.7.2	异步读取数据使用示例	562
20.8	本章总结	564
20.9	实战练习	564
第 21 章	可扩展标记语言 (XML) 教学视频: 30 分钟	565
21.1	认识 XML	565
21.1.1	文档对象模型的功能	566
21.1.2	用 XPath 查询 XML 文档	567
21.1.3	了解可扩展样式表语言 XSL	567
21.1.4	用 XML Schemas 设置数据元素和属性	568
21.1.5	.NET 中处理 XML 的相关类	568
21.2	使用 XML DOM 进行编程	569
21.2.1	创建一个空的 XML 文档	571
21.2.2	向 XML 文档添加元素	572
21.2.3	更新 XML 文档中的元素	573
21.2.4	删除 XML 文档中的元素	576
21.2.5	加载和保存 XML 文档	577
21.3	用 DataSet 保存 XML 数据	577
21.3.1	不使用 Schema 文件加载 XML 文档	580
21.3.2	使用 Schema 文件加载 XML 文档	582
21.3.3	遍历 XML 文档	584
21.4	用 XPath 查找节点	586
21.4.1	XPath 简介	586
21.4.2	XPath 查询示例代码	586
21.4.3	XPath 示例代码讲解	590
21.5	使用 XSL 将 XML 文档转化为另一种格式	591
21.5.1	XSL 转换示例一	591
21.5.2	XSL 转换示例讲解	595
21.5.3	XSL 转换示例二	596


21.6	本章总结	597
21.7	实战练习	597
第 22 章	多线程编程 (🔑 教学视频: 27 分钟)	599
22.1	C#线程的主要特征	599
22.1.1	输出不同内容的两个线程	599
22.1.2	调用同一方法的两个线程	600
22.1.3	静态变量在多线程多线程中的应用	601
22.1.4	线程调度	603
22.1.5	线程和进程的关系	603
22.1.6	何时需要使用多线程	603
22.1.7	何时不要使用多线程	604
22.2	创建并开始一个线程	604
22.2.1	用 Thread 类创建线程	604
22.2.2	向 ThreadStart 传递参数	605
22.2.3	给线程命名	607
22.2.4	C#的后台线程	607
22.2.5	设置线程优先级	608
22.2.6	线程中的异常处理	608
22.3	线程同步	610
22.3.1	线程同步和协同的可用性工具	610
22.3.2	阻止现在的线程	611
22.3.3	Joining 一个线程	612
22.4	线程安全	612
22.4.1	了解线程安全	612
22.4.2	选择一个同步对象	614
22.4.3	使用嵌套锁	615
22.4.4	什么时候上锁合适	615
22.4.5	使用锁的效率考虑	616
22.4.6	线程安全与.NET Framework	617
22.5	用中断和取消提前释放线程	618
22.5.1	中断线程	618
22.5.2	取消线程	619
22.6	线程有哪些状态	619
22.7	等待处理	620
22.7.1	了解自动设置方法	621
22.7.2	自动设置方法示例一	621
22.7.3	自动设置方法示例二	622
22.7.4	自动设置方法示例三	623
22.7.5	ManualResetEvent 类控制多个线程	625
22.7.6	跨线程的互斥量	626

22.7.7 使用信号量	626
22.7.8 使用 WaitAny、WaitAll 和 SignalAndWait 方法	627
22.8 同步性作用域	628
22.9 套间线程	630
22.9.1 什么是套间线程	630
22.9.2 使用套间模型	631
22.9.3 用 Control.Invoke 方法进行跨线程调用	631
22.10 管理工作线程的 BackgroundWorker 组件	631
22.10.1 BackgroundWorker 的特征	632
22.10.2 BackgroundWorker 组件编程示例一	632
22.10.3 BackgroundWorker 组件编程示例二	633
22.10.4 BackgroundWorker 组件编程示例三	635
22.11 用于读/写操作的锁	636
22.11.1 了解读/写操作的锁	636
22.11.2 管理资源访问锁定状态类 ReaderWriterLockSlim	637
22.11.3 读/写操作锁的进一步说明	638
22.12 用线程池管理线程	639
22.13 用异步代理得到线程返回的数据	641
22.14 .NET 提供的计时器	642
22.15 各线程数据的局部存储	644
22.16 本章总结	644
22.17 实战练习	645


第 6 篇 Web 数据库开发



第 23 章 数据库基础知识 ( 教学视频: 15 分钟)	648
23.1 了解 SQL Server	648
23.2 操作 MSSQL 数据表	649
23.2.1 在数据库中创建、修改和删除表	650
23.2.2 向表中插入、修改、删除和检索数据	650
23.2.3 设置表的主键约束	653
23.2.4 设置表的外键约束	654
23.2.5 设置表的唯一性约束	654
23.2.6 设置表的 CHECK 约束	655
23.2.7 设置列的默认约束	655
23.3 数据库的存储过程	655
23.3.1 创建存储过程的 SQL 语句	656
23.3.2 执行和删除存储过程的 SQL 语句	657
23.3.3 用 SQL Server Management Studio 管理存储过程	657

23.4	数据库中的触发器	659
23.4.1	创建和使用触发器的 SQL 语句	659
23.4.2	用 SQL Server Management Studio 管理触发器	660
23.5	本章总结	661
23.6	实战练习	661
第 24 章	ADO.NET 数据库编程 ( 教学视频: 32 分钟)	662
24.1	ADO.NET 介绍	662
24.1.1	ADO.NET 是神马	662
24.1.2	ADO.NET 相关的类和接口	663
24.2	DataSet 和 DataTable 类	664
24.2.1	表示内存数据表的 DataTable 类	664
24.2.2	创建 DataTable 的方法	666
24.2.3	遍历 DataTable 中保存的记录	669
24.2.4	接受和回滚 DataTable 的更改	670
24.2.5	表示内存数据集合的 DataSet 类	673
24.2.6	使用 DataSet 类的步骤	675
24.2.7	接受和回滚 DataSet 的更改	679
24.3	用 ADO.NET 访问 SQL Server 数据库	680
24.3.1	ADO.NET 访问数据库的步骤	680
24.3.2	用 SqlConnection 连接数据库	681
24.3.3	用 SqlCommand 执行 SQL 命令	684
24.3.4	用 SqlDataReader 读取数据库记录	687
24.3.5	用 SqlDataAdapter 获取数据库记录	689
24.3.6	用 SqlDataAdapter 更改数据库记录	692
24.4	用 ADO.NET 访问 Access 数据库	694
24.4.1	System.Data.OleDb 命名空间提供的功能	694
24.4.2	访问 Access 数据库的各种类	695
24.5	使用数据库访问控件	696
24.5.1	用 DataGridView 控件管理数据库中的记录	696
24.5.2	用 BindingNavigator 控件导航记录	699
24.6	本章总结	700
24.7	实战练习	700
第 25 章	ASP.NET 技术入门 ( 教学视频: 24 分钟)	702
25.1	初识 ASP.NET	702
25.1.1	了解 ASP.NET	702
25.1.2	System.Web 常用的类	703
25.1.3	创建一个 Web 应用程序	703
25.2	用 ASP.NET 控件创建网页	706
25.2.1	用 TextBox 控件显示文本框	706

25.2.2	用 Button 控件显示按钮	707
25.2.3	用 HyperLink 控件显示超链接	709
25.2.4	用 DropDownList、ListBox 等显示下拉列表、列表	709
25.2.5	用 Menu 控件显示导航菜单	711
25.3	留言板网站实例	713
25.3.1	数据库和页面设计	713
25.3.2	设计欢迎页面	714
25.3.3	设计添加留言页面	715
25.3.4	设计查看留言页面	718
25.3.5	发布留言板网站	721
25.4	本章总结	723
25.5	实战练习	724
第 26 章	服务器端控件详解 ( 教学视频: 24 分钟)	725
26.1	认识服务器控件	725
26.1.1	为什么使用服务器控件	725
26.1.2	服务器控件与 HTML 控件的区别	725
26.2	数据操作控件	726
26.2.1	SqlDataSource 的作用	726
26.2.2	用 SqlDataSource 控件连接到数据库	727
26.2.3	用 GridView 控件显示数据	732
26.2.4	用 DetailsView 控件显示指定记录	735
26.3	用验证控件检查用户输入	739
26.3.1	必填内容的验证控件 RequiredFieldValidator	739
26.3.2	比较两个值的验证控件 CompareValidator	740
26.3.3	检查指定范围的验证控件 RangeValidator	743
26.3.4	正则表达式验证控件 RegularExpressionValidator	743
26.3.5	自定义验证控件 CustomValidator	744
26.3.6	验证错误信息汇总控件 ValidationSummary	747
26.4	创建 ASP.NET 用户控件	748
26.5	本章总结	751
26.6	实战练习	751

第 7 篇 .NET 4.0 的增强功能

第 27 章	WPF 框架 ( 教学视频: 19 分钟)	754
27.1	WPF 基础	754
27.1.1	了解 WPF 基础架构	754
27.1.2	与 WPF 相关的技术	755

27.2 创建 WPF 应用程序.....	756
27.2.1 创建 WPF 的过程.....	756
27.2.2 完整的 WPF 应用程序实例.....	759
27.2.3 创建 WPF 浏览器应用程序.....	761
27.3 简单 WPF 实例.....	763
27.3.1 用 ListBox 控件实现列表显示.....	763
27.3.2 用 Hyperlink 控件实现多页面切换.....	767
27.3.3 用 DockPanel 沿容器边缘定位.....	768
27.3.4 使用 StackPanel 叠放包含的控件.....	770
27.3.5 使用数据源集合实现多数据绑定.....	771
27.3.6 属性变更引起依赖数据绑定变化.....	774
27.3.7 使用 Brush 填充图形.....	778
27.3.8 使用 Storyboard 实现动画.....	783
27.3.9 使用 Storyboard 实现控件的翻转.....	785
27.4 本章总结.....	787
27.5 实战练习.....	787
第 28 章 WCF 框架 ( 教学视频: 22 分钟)	788
28.1 WCF 基础.....	788
28.1.1 了解 WCF 架构.....	788
28.1.2 了解 WCF 模型.....	789
28.2 WCF 服务和客户端实例.....	790
28.2.1 创建 WCF 服务承载项目.....	790
28.2.2 定义 WCF 服务协定.....	790
28.2.3 定义实现 WCF 服务接口的类.....	792
28.2.4 运行 WCF 服务的相关代码.....	793
28.2.5 创建 WCF 客户端程序.....	795
28.2.6 配置 WCF 客户端的配置.....	795
28.2.7 WCF 客户端对服务端的调用.....	797
28.3 主要的 WCF 技术.....	799
28.3.1 使用会话在客户端与服务间交互.....	799
28.3.2 WCF 事务管理模型.....	800
28.4 本章总结.....	801
28.5 实战练习.....	802
第 29 章 Windows WF 框架 ( 教学视频: 40 分钟)	803
29.1 C#的工作流开发框架.....	803
29.1.1 了解 WF 框架.....	803
29.1.2 WF 框架中的重要元素.....	804
29.2 开发 WF workflow 应用程序.....	805
29.2.1 第一个 WF 应用程序.....	805

29.2.2	WF 工作流的基本元素: WF 活动	810
29.2.3	自定义的代码活动	815
29.2.4	WF 提供的服务	818
29.3	WF 创建工作流实例	819
29.3.1	在工作流中使用集合	819
29.3.2	猜价格游戏	823
29.4	本章总结	828
29.5	实战练习	828
第 30 章	语言集成查询 LINQ ( 教学视频: 25 分钟)	829
30.1	LINQ 概述	829
30.1.1	了解 LINQ 查询	829
30.1.2	简单 LINQ 查询实例	830
30.2	LINQ 语言基础	830
30.2.1	最重要的 LINQ 查询表达式	831
30.2.2	LINQ 查询语法和方法语法实例	833
30.2.3	用 LINQ 合并数据	835
30.2.4	用 LINQ 转换数据	837
30.3	LINQ 查询数据源	838
30.3.1	用 LINQ To SQL 查询数据库中的数据	838
30.3.2	用 LINQ To DataSet 查询缓存在 DataSet 中的数据	842
30.3.3	用 LINQ To XML 查询 XML 中的数据	843
30.3.4	用 LINQ To Objects 查询可枚举的集合	844
30.4	本章总结	846
30.5	实战练习	846

*第 8 篇 综合案例

第 31 章	用 MVC 开发音乐商店网站	848
31.1	开发站点前的配置	848
31.1.1	预览音乐商店网站	848
31.1.2	系统架构总览	851
31.1.3	系统数据库设计	852
31.1.4	系统文件目录结构	854
31.1.5	创建母版页和 CSS 文件	856
31.1.6	使用 Entity Framework 创建实体模型	858
31.2	音乐专辑列表实现	860
31.2.1	首页控制器实现	860
31.2.2	创建首页视图	861
31.2.3	浏览音乐流派	862
31.2.4	浏览音乐明细信息	865

31.3	管理音乐列表	866
31.3.1	基于角色的身份验证	867
31.3.2	编辑音乐信息	870
31.3.3	基于模型的数据验证	875
31.4	实现和管理购物车	876
31.4.1	添加到购物车功能	876
31.4.2	从购物车中移除功能	879
31.4.3	提交购物车	880
31.5	关键技术讲解	885
31.6	本章总结	885
31.7	实战练习	885
第 32 章	电子购物商城	886
32.1	系统总体设计	886
32.1.1	系统功能描述	886
32.1.2	WebShopping 应用程序组成	888
32.1.3	保存数据需要哪些表	890
32.1.4	设计数据库表之间关系设计	893
32.2	系统通用类和模块设计	894
32.2.1	编写系统常量和枚举代码	894
32.2.2	编写系统常用操作的代码	895
32.2.3	设计显示标题的用户控件	896
32.2.4	设计分页用户控件	897
32.2.5	创建级联样式表 web.css 文件	902
32.2.6	设计主题文件	902
32.2.7	在 Web.config 中设置数据库连接字符串	903
32.2.8	创建数据库对象模型	904
32.3	电子购物商城首页	907
32.3.1	设计首页界面	907
32.3.2	设计全站搜索的用户控件	911
32.3.3	设计登录网站的用户控件	912
32.3.4	设计显示新闻的用户控件	914
32.3.5	设计显示公告的用户控件	916
32.4	用户注册和登录	917
32.4.1	用户必须先注册	918
32.4.2	显示提交的用户信息	922
32.4.3	处理用户登录	925
32.4.4	退出系统时要做的操作	928
32.5	浏览商品、购物车和订单	928
32.5.1	查看商品信息	929
32.5.2	对商品进行评论	932
32.5.3	购物车功能	936
32.5.4	将商品加入购物车	937
32.5.5	查看购物车内容	940
32.5.6	生成订单编号的代码	944
32.5.7	生成购物订单	945

32.5.8	发布留言	950
32.6	商品搜索	952
32.6.1	按商品名称搜索	953
32.6.2	更复杂的高级搜索	956
32.7	用户信息中心	960
32.7.1	管理用户的首页	960
32.7.2	根据角色动态加载操作菜单	962
32.7.3	查看个人信息	965
32.7.4	修改个人信息	967
32.7.5	修改密码	971
32.7.6	管理我的订单	974
32.8	商品及其分类管理	978
32.8.1	查看已有商品分类列表	978
32.8.2	动态加载商品分类层次结构的代码	983
32.8.3	添加新的商品分类	984
32.8.4	修改已有商品分类	987
32.8.5	查看已有商品列表	990
32.8.6	动态加载商品分类层次树的代码	997
32.8.7	添加新的商品	999
32.8.8	修改已有商品	1003
32.8.9	商品图片管理	1007
32.8.10	显示商品图片	1012
32.8.11	商品评论列表	1013
32.9	系统信息中心	1018
32.9.1	查看已有新闻列表	1018
32.9.2	发布新的新闻	1023
32.9.3	修改已发布新闻	1025
32.9.4	发布公告	1029
32.9.5	管理用户留言	1032
32.10	系统用户和角色管理	1035
32.10.1	查看已有角色列表	1036
32.10.2	添加新的角色	1040
32.10.3	修改已有角色	1042
32.10.4	管理注册用户	1045
32.10.5	添加管理员	1049
32.11	本章总结	1053
32.12	实战练习	1053

说明：因篇幅所限，第8篇内容以PDF电子文档的格式收录于本书的配书光盘中。该项目案例涉及的源程序也收录于配书光盘中。

第 1 篇 开发环境及 C# 语言简介

- ▶▶ 第 1 章 Visual Studio 2010 介绍
- ▶▶ 第 2 章 C#简介

第 1 章 Visual Studio 2010 介绍

Visual Studio 2010 是 Visual Studio .NET 家族的最新成员。在介绍 Visual Studio 2010 之前，要先从 Visual Studio .NET 说起。

什么是 Visual Studio .NET 呢？它是一套完整的开发工具，可以用于生成桌面应用程序、ASP Web 应用程序、XML Web 服务，以及移动应用程序等。在 Visual Studio .NET 环境下，Visual Basic .NET、Visual C++ .NET 和 Visual C# .NET 等开发技术全都使用相同的集成开发环境，即 Visual Studio .NET IDE。从 Visual Studio .NET 2002 开始，微软将 Visual J++ 改版为 Visual J#，这主要是为了支持 Java 语言而设置的。到 Visual Studio 2008 时取消支持 Visual J#。在 Visual Studio 2010 中，新支持 Visual F#。有关 Visual Studio 2010 的独特功能，将在本章中介绍。

Visual Studio 2010 环境允许所有的语言共享工具，这样有助于创建混合语言解决方案。这些语言利用了 .NET Framework 的所有功能。Visual Studio 2010 可以用于访问简化 ASP Web 应用程序，以及 XML Web Services 开发的关键技术。

本章将带领读者一起来揭开 Visual Studio.NET 的神秘面纱，试着了解这套风靡全球的软件开发工具的实质，了解其真正的魅力所在。

1.1 .NET 概述

2000 年 6 月，Microsoft 公司总裁比尔·盖茨先生在一次名为“论坛 2000”的会议上发表演讲，首次提出了 .NET 框架的理念，并且为与会者描绘了 .NET 的美丽前景。从那一天起，整个 IT 界的 .NET 时代到来了。

1.1.1 .NET 的前世今生

在 Microsoft 公司的强力推动下，.NET 从无到有最终成为一种先进的集成软件开发工具。其发展史基本可以总结如下：

- ❑ 2002 年 1 月，Microsoft 公司首先公布了 .NET Framework 1.0 正式版，并同步发行了 Visual Studio 2002.NET。
- ❑ 2003 年 4 月 23 日，Microsoft 公司又推出了 .NET Framework 1.1 和 Visual Studio 2003.NET。从功能上来看，这两项产品都是针对 .NET 1.0 的升级版本。
- ❑ 2004 年 6 月，Microsoft 发布了 .NET Framework 2.0 Beta1 和 Visual Studio 2005 Beta1 两项产品，与此同时还发布了多个 Express Edition 精简版，其中包括著名的 Visual

Web Developer 2005、Visual Basic 2005、Visual C# 2005 和 SQL Server 2005 Express Edition 等。此时，Visual Studio.NET 产品已经成为了 IT 界的大趋势。

- ❑ 在业内外人士的期盼与等待下，Microsoft 公司终于在 2005 年 4 月发布了 Visual Studio 2005 Beta2 测试版。
- ❑ 2005 年 11 月，Microsoft 公司又发布了 Visual Studio 2005 和 SQL Server 2005 正式版。
- ❑ 此后，Microsoft 公司曾经有意研究出 Visual Studio 2005 的升级产品 Visual Studio 2007，但一直没有正式发布公开版本。
- ❑ 2007 年 7 月，在万众瞩目之下，Visual Studio 2008 Beta2 和 .NET Framework 3.5 Beta2 横空出世，新一轮的 Visual Studio .NET 旋风再度席卷全球。
- ❑ 2010 年 4 月，微软公司推出了 Visual Studio 2010，其集成开发环境（IDE）的界面被重新设计和组织，变得更加简单明了。Visual Studio 2010 同时带来了 .NET Framework 4.0、Microsoft Visual Studio 2010 CTP（Community Technology Preview，CTP），并且支持开发面向 Windows 7 的应用程序。除了 Microsoft SQL Server，它还支持 IBM DB2 和 Oracle 等数据库。

1.1.2 什么是微软中间语言

在 .NET 框架中，公共语言运行时（Common Language Runtime）使用公共语言规范来绑定不同的语言。公共语言运行时允许不同的语言使用 .NET 框架。只要不同的语言实现了公共类型系统包含在公共语言规范中的部分，所有的语言均可在 .NET Framework 中运行。因此在 .NET 框架中，所有的语言在编译时都被转换为了一种通用语言，即微软中间语言（Microsoft Intermediate Language，MSIL）。

微软中间语言是将 .NET 代码转化为机器语言的一个中间过程，它是一种基于 Intel 汇编语言的伪汇编语言。由于介于高级语言和汇编语言当中，中间语言的语法简单，它使用数字代码进行表述，以便可以非常快速地转化为机器内部代码。这就使得所有 .NET 语言编译器生成与平台无关的代码，当用户编译一个 .NET 程序时，编译器将源代码翻译成一组可以有效地转换为本机代码且独立于中央处理器（CPU）的指令。在通常状况下，习惯将 MSIL 简称为 IL。

在运行过程中，当第一次调用一个方法时，为了能更快地执行程序，实时 Just-In-Time（JIT）编译器会将该方法的 IL 代码转换成源代码，这个过程与平台有关。.NET 只编译需要在运行库中使用的 IL，转换后的源代码被放入缓存中。当编译器产生 MSIL 的同时，也产生了元数据。

元数据中包含了很多的综合信息，它们描述了代码中每一种类型的定义、每一种类型成员的签名、代码引用的成员，以及在运行时用到的其他数据。元数据和 MSIL 同时产生存在，在它的辅助下代码拥有了自描述的特性，同时也就不再需要类型库或者 IDL。在执行过程中，运行是根据需要从文件中定位和提取元数据。这样的一个过程确保了中间语言的独立性。

📢说明：从理论上说，MSIL 消除了多年以来不同语言之间的纷争。这给.NET 用户提供了极大的便利，用户可以选择自己熟悉的语言，再也不用为学习不断推出的新语言而烦恼了。

1.1.3 背后默默付出的垃圾收集器

垃圾收集器（Garbage Collector，GC）是一种虚拟机技术，它最早出现于 Java 语言中。

在介绍垃圾收集器之前，首先有必要先了解一下垃圾收集机制。在面向对象的编程语言中，垃圾收集机制首先见于 Java 语言。对于垃圾收集器而言，从程序员创建对象伊始，它就开始执行对这个对象的监控。主要监控内容包括对象的地址、大小，以及使用情况。利用垃圾收集器，程序可以自动确定不再被利用的对象，并自动将它删除。

内存管理是计算机内部职责的一种，当程序员捕捉到记忆体泄露时，它主要负责将运行的程序从正在开发的代码中移出。要知道，有时需要花费一天的时间去解决一个复杂难懂的内存问题，这是非常没有效率的事情。

通常，垃圾收集器会利用有向图的方式记录并管理存储在堆（heap）中的所有对象。通过这种方式，垃圾收集器可以确定哪些对象可用，哪些对象不可用。当垃圾收集器确定某些对象为不可用时，它就回收这些内存空间。

Java 利用垃圾收集器的内存管理方式是其重要的一大优势。因此，Visual Studio .NET 发布伊始便借鉴了这种内存管理方法，并根据.NET 自身的特点做出了相应的改进，使得垃圾处理器的处理效率有了一定的提高。

.NET 希望能够实现某种垃圾收集器的系统，那样就可以在托管的环境下帮助程序员解决这类问题。当程序员的应用程序已经明显超过空闲内存时，或者当应用程序被间接地调用却不能立即执行而无端占用内存的时候，垃圾收集器就会运行。

现在先来分析一下系统是如何运作的吧。当应用程序申请占用了过多内存的时候，内存分配符就会报告情况，提示托管堆中已经没有更多的内存空间了，此时垃圾收集器被自动调用。垃圾收集器开始工作以后，首先将内存中所有的东西都假定为能被释放的，然后垃圾收集器会遍历应用程序的内存，并且建立一个所有关于这个应用程序所引用的当前资源的分析曲线图。一旦垃圾收集器完成了曲线分析图，它将会移动所有正在空闲的内存堆的开始部分集体运作的内存资源，以精简堆的压力。所有这些完成以后，垃圾收集器会将指针收回，这样一来内存分配符就可以发挥功用了。内存分配符通常会从这个崭新的堆的顶部开始决定如何重新分配内存资源。


除了这些工作以外，垃圾收集器也可以更新应用程序引用资源，这样就可以为它们指出被分配在内存中的新地址了。这种方法通常被用于标记或者清除占用资源。当垃圾收集器处理一个堆的资源时，一些比较大的对象从其他的堆中被分派过来，但是它们并没有被移动，因为这种大体积的可移动内存资源将会对程序的运行产生负面影响。

正如读者所想的，垃圾收集器囊括了许多工作，当然也就占用了一些时间。通常情况下，在需要的时候可以借助 CLR 来控制垃圾收集器的运行。然而，程序员往往希望能够自主控制垃圾收集器，尽管这种行为是不被提倡的，但是程序员仍然可以通过调用 `GC.Collect()` 函数来实现在开始程序前请求大额度内存的操作。如果想要在程序运行时通过内存报告来监测何时启动强制垃圾收集，`GC.GetTotalMemory (bool forceFullCollection)` 函数就可以

帮忙了。

.NET 平台提供的垃圾收集器可以自动承担回收为对象分配的内存,这就是所谓的托管堆。.NET 运行时托管资源 (Managed Resource) 采用堆式分配 (Heap Allocation) 方法,这使得内存的处理速度大大加快。当系统剩余的可分配的内存资源减少到一定的边界域值时,.NET 运行时便会察觉到内存资源可能不满足内存分配的请求,于是它便会开始执行垃圾回收,以释放一些系统不再引用的内存资源。这个过程实现了垃圾收集的机制。

.NET 中,垃圾收集器采用的是一种被叫做标志紧缩 (Mark and Compact) 的算法。当垃圾收集器开始,.NET 垃圾收集器从运行时目前的根对象,包括全局对象、本地对象、静态对象和 CPU 寄存器对象等,开始寻找那些被根对象引用的所有对象。这些被引用的对象便是垃圾收集运行时正在应用的对象,除此之外的所有其他托管运行时对象 (Managed Runtime Object) 便是系统不再引用的对象。这种算法判断之后,垃圾收集工作就开始了。

 **说明:** 垃圾收集器是在.NET 程序调用析构函数的时候开始自动执行的。这些都将由运行时的互斥器来保证,程序员不用手动进行。

关于垃圾收集器的具体实现,本书将在后面的章节中举例详细介绍。

1.2 .NET Framework 概述

1.2.1 .NET Framework 包含什么

想要真正了解 Visual Studio.NET,就必须先知道何谓.NET Framework。.NET Framework 是一个由 Microsoft 提供的完善且透明的基础架构,是一个开发平台,是一组封装好的有效程序集合。在 Visual Studio 中,.NET Framework 作为一个对象的集合,直接存在于.NET 的框架中。用户在需要的时候可以不必再亲自撰写一些类似的重复性代码,取而代之的是直接调用。

.NET Framework 是整个.NET 平台的基础结构,是用于生成、部署和运行 XML Web Services,以及其他应用程序的环境。它包含一个庞大、有效、高度封装的代码库,这使得.NET Framework 可以在用户使用某种语言时,通过面向对象编程技术 (OOP) 来复用这些代码。依据用户的实际需要,.NET Framework 的代码库被分为了诸多不同的模块。这些模块实现了不同的开发功能需求,方便了用户在开发时可以选择性地使用代码库的相应部分。这样不仅可以提高.NET Framework 环境的服务效率,而且便于其升级开发。

任何一项技术或者一种概念的发展历程都是充满了争议与分歧的,.NET 当然也不例外。关于.NET Framework 的构成划分而言,一直存在着很多的不同意见。主流而言,可以将其分为 4 层来理解,即应用程序、Framework 框架类库 (FCL)、基类库 (BCL) 和公共语言运行时 (CLR)。也有人将.NET Framework 简单地划分为两个部分,即公共语言运行时 (Common Language Runtime, CLR) 和 Framework 框架类库 (Framework Class Library, FCL)。

本书主要参考了.NET Framework 的工作原理和结构构成的现实意义,将.NET Framework 划分为以下 3 个主要部分:

- 公共语言运行时 (Common Language Runtime, CLR);

❑ 基类库 (Base Class Library, BCL) ;

❑ Web Applications (ASP.NET), 以及常规 Windows Applications (Windows Forms)。

除了这些主要的组成部分之外, .NET Framework 还包括一些开发周期必备条件。例如, 代码复用、精简编程支持、简易开发、垃圾收集和中间语言等。所有的这些共同构成了 .NET Framework 的完备支持条件。图 1.1 直观地展示了 Visual Studio .NET 的构成。

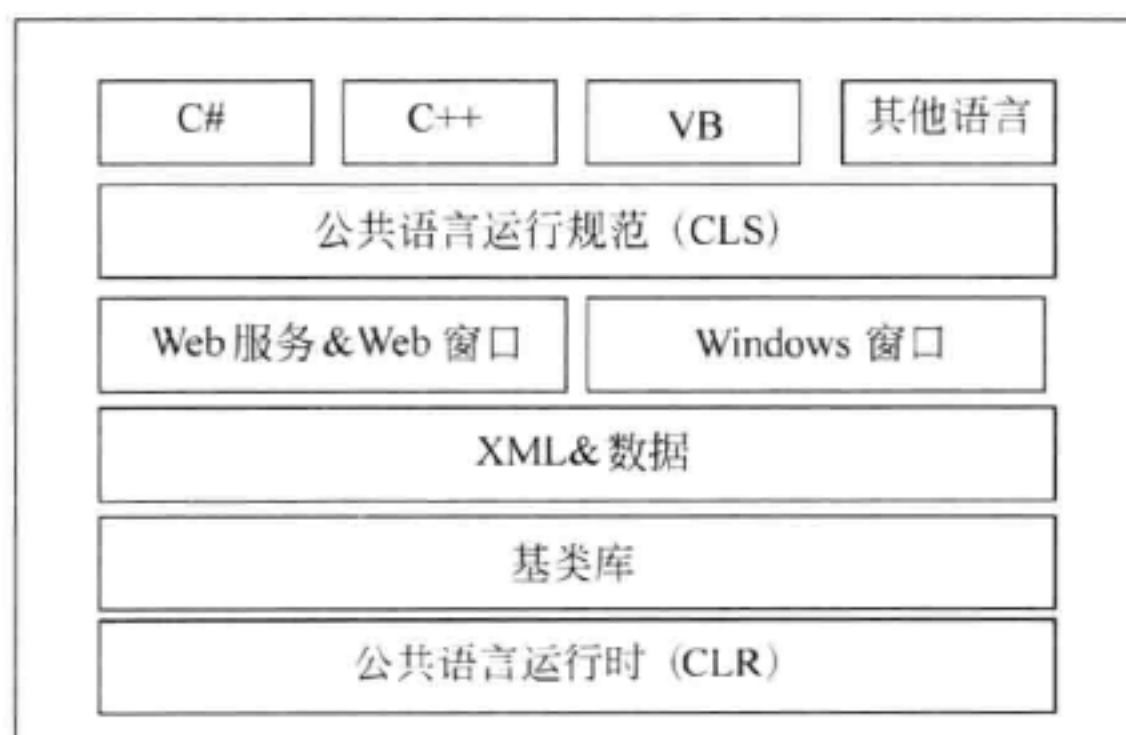


图 1.1 Visual Studio .NET 组成

注意：NET Framework 对很多的底层技术进行了大量的抽象，是为平台或 Windows 操作系统本身提供的一种技术。这使得用户能够更多地思考如何解决他们自己的项目问题，而不是去思考机器和操作系统本身的问题。

实际上，在 .NET Framework 平台中公共语言运行时和基类库是结合在一起的。它们的结合使得 .NET Framework 异常强大，能够为用户提供在各种系统内部的服务和解决方案。此外，.NET Framework 也支持各系统之间的兼容与集成。.NET Framework 提供完全托管的、受保护的，并且功能丰富的应用程序执行环境、简化的开发和部署，以及与各种语言的无缝集成。

使用 .NET Framework 编写应用程序，就是使用 .NET 代码库编写代码。在编译使用 .NET Framework 库的代码时，后台操作并不是立即创建操作系统特定的本机代码，而是把这些代码编译为 Microsoft 中间语言 (Microsoft Intermediate Language, MSIL) 代码。这些 MSIL 代码不专用于任何一种操作系统，也不专用于 .NET 的任何特定语言，只要在 .NET Framework 的支持下，任何 .NET 语言都可以在第一阶段将需要执行操作的代码编译为这种语言。

.NET Framework 为使用者提供了一个一致的面向对象的编程环境，它最突出的贡献在于代码执行安全性的提高，以及代码执行环境的构建。在 .NET Framework 平台上，代码执行所需的脚本环境或解释环境的性能得到了充分的保证。因此，.NET 开发的程序需要在 .NET Framework 下才能运行。通过 CLR 为基础，支持多种语言，如 C#、VB (Visual Basic)、C++ 和 F# 等的开发。

根据各个部分的不同功能，.NET Framework 真正强调了代码重用、代码专用、资源管理、多语言开发、安全性、部署，以及管理问题。因而在简化编程模型、简化部署、编程语言集成、自动内存管理 (垃圾自动回收)、类型安全验证，以及互操作性等方面都有了出色的表现。在接下来的各个小节中，本书将着重为读者介绍 Visual Studio .NET 的各个组成部分的意义。


1.2.2 .NET 平台的核心：CLR

在 1.2.1 节中提到了 CLR，即公共语言运行时（Common Language Runtime），它的前身是 COM3，是 Microsoft 公司的 COM 与 MTS 小组计划开发一个新的组件平台。就其本质而言，它是编译器对托管代码进行操作控制的最关键所在。

CLR 是一个托管的处理环境，用以处理存储配置、错误跟踪，以及与操作系统服务的信息交互，它其实是一个虚拟执行系统，主要负责运行所有的托管代码。

要了解其意义就必须知道其工作原理。在 .NET Framework 平台下，将高级语言程序编译成机器指令需两个步骤。首先，程序被编译成 Microsoft 中间语言（Microsoft Intermediate Language, MSIL），MSIL 定义了 CLR 指令。.NET Framework 支持包括 C# 在内的多种计算机语言的编译，从其他语言转换成 MSIL 的代码在公共语言运行时被整合在一起。然后，CLR 的另一个编译器可以创建一个单独的应用程序，并且能够将 MSIL 编译成针对指定平台的机器代码。由于不同操作系统间的可移植性不同，而语言之间的互操作性也不一样，这就需要公共语言运行时来处理这些差异所带来的问题。

CLR 以规范的格式来描述组件之间的约定，这种格式一般称为元数据。编译器执行程序时，伴随着源代码转换为中间代码也将自动生成元数据。元数据携带了源代码中类型信息的描述。这些元数据与编译后的源代码共同被包含在二进制代码文件中。从本质上而言，.NET 家族的所有语言都符合 MSIL 的输出条件，而 MSIL 是依照公共语言运行时定义的。.NET Framework 平台中主要的应用开发程序种类包括 Web 窗口、Web 服务、Windows 窗口应用程序等。这些应用程序利用 XML，以及简单对象访问协议（Simple Object Access Protocol, SOAP）从基类库中获得功能性函数，并且在运行公共语言运行时的环境下才能达到预定的效果。由于元数据携带了应用程序的类型信息，所以系统通过 CLR 装载类型时，那些附带在元数据上的类型信息将会被 JIT（Just-In-Time）编译器分析加载。然后，编译器就能将中间语言代码转化成为本地代码，在此基础上根据程序或用户要求建立类型的实例。由于整个过程中，CLR 始终根据元数据建立并管理对应特定应用程序的类型，从而保证了类型安全性。

 **说明：**公共语言运行时是 .NET 平台的核心，这如同 Java 中的虚拟机一样。CLR 是一种运行时环境，用以执行 MSIL 代码。

不同于 Java 环境的是，作为一种多用途语言的关键概念，.NET 平台支持多种计算机语言共同使用公共语言运行规范（CLS）。

公共语言运行规范使得基于 Visual Studio.Net 开发的应用程序的跨语言交互成为可能。CLS 定义了基于 .NET Framework 类库中的类的基本组成成员的类型子集，并且建立了 CLS 的遵从性要求，使得任何符合 CLS 规范的程序组件都可以被其他符合 CLS 的应用程序访问。

1.2.3 基类库是神马

当一个程序员开始编程的时候，如果他可以拥有一间图书馆，而它所提供的一切都是 .NET 基类库（Base Class Library, BCL）所包含的，那么付给足够的薪水也是非常合理

的。在.NET 的基类库中几乎可以包含一切程序员所必需的基础功能封装。几乎.NET 环境里的所有事物都被包含在其基类库(BCL)里。下面以一个C#程序中最简单的Hello World程序为例,来讲解基类库的作用与使用。


```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//声明此程序的命名空间
namespace _1._1
{
    //类Hello 声明
    class Hello
    {
        //程序入口
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

被包含在这个简单程序中唯一的函数就是对控制台类 Console 中的 WriteLine 方法的调用。.NET 环境的真正独特之处在于.NET 语言不需要去实现那些最基本的函数,这些工作都已经由 BCL 完成了。由于所有的.NET 语言都共享用相同的公共类的设置,所以 C# 的程序代码在运行时的处理等同于其他任何一种代码的运行处理。这就意味着所有的语言都以.NET 环境为目标,除了它们有不同的语法之外,本质上它们共同利用着.NET Framework 的处理能力。

讲到这里,有的读者可能想要知道:既然这些不同的计算机语言运用同一种处理能力,那么为什么要区分不同的语言种类呢?主要的理由如下:

- ☐ 不同语言的程序员习惯固定的语法,不想改变。
- ☐ 程序员通常会精通一种最爱的语言。
- ☐ 程序员不习惯改变。

想象一下,假如 Microsoft 公司突然宣布,已经将所有优秀的技术都集成到.NET 中,但是前提条件是大多数程序员需要重新学习使用一门新的语言才能够使用这些技术,那么程序员们将会做何选择呢?相信大多数的人就不会将信任给予.NET 了,除非是作为雇员而不得已使用。所以,将.NET 的可用性包含在多种计算机语言之中可以令它受到更大程度的接受以及欢迎,程序员不需要重新学习一门新的技术就可以很快地熟悉使用.NET,并且会十分欢迎 BCL——这个将所有的基础函数集成的类库。正是由于 BCL 的存在才使得.NET 平台下的编程工作变得更加简单。

 **说明:** 基类库本质上是一个关于编程控件,以及应用程序编程接口(Application Programming Interface, API)的广泛集成。

几乎所有的.NET 语言都可以直接调用 Win32 API。从理论上来讲,所有.NET Framework 中的语言都是有平等的机会去调用 BCL 的基础方法函数的。然而,实际上,不同的语言在语言队列中对 BCL 的支持程度是有所差异的。这点将会在以后各章中介绍 C#


特性的时候为读者进一步详细说明。

当 Windows API 允许用户调用 Windows 操作系统的特征元素的时候，.NET 的基类库会通过相同的方式展现公共语言运行时的特征。当然，基类库也可以提供一些可以方便代码复用的更高层次的元素特征。而公共语言运行规范给予语言研发者，以及开发人员最基本的基础需求，这些基本规范都为代码能够实现公共语言运行时而服务。.NET 平台允许语言相互整合，通过利用 MSIL 使得所有语言最终的输出都能满足平台设计需求。

之前已经提到了 .NET Framework 的划分方法中一个比较重要的概念——框架类库（Framework Class Library, FCL）。本质上而言，框架类库（FCL）是 BCL 的一个重要的组成部分。由于 .NET 是 Microsoft 公司在其 Windows 平台下开发出来的产品，所以 .NET 中其实已经包含了很多 Windows 本身的服务，以及类库信息。有些业内的学者甚至认为，除 ASP.NET 和 .NET Framework 以外，.NET 的所有服务相关的支持都是基于 Windows Server Family 的。因此，很多的资料将 BCL 等同于 FCL，这实际上是将 BCL 的概念缩小了。它们最大的共同特点就是，都是基于公共语言运行时（CLR）以托管代码为特征的所有应用。了解了 BCL 的定义与工作原理之后，读者也就不难掌握 FCL 的相关知识了。这里就不再赘述。

1.2.4 公共语言运行规范

实际上，.NET Framework 由公共语言运行时、基类库，以及公共语言运行规范 3 个主要部分组成。公共语言运行规范（Common Language Specification, CLS）描述了使用其他开发工具的程序员必须执行的规范与步骤。这些规范是为了令那些编译器可以输出符合中间语言的代码，这些中间代码将会利用其他的 .NET 语言把各种编译器环境很好地整合起来。必须要感谢公共语言运行规范（CLS）及公共类型系统（Common Type System），正是由于有了它们的支持，才使得 .NET 实现了跨组件，以及跨语言的工作能够顺利完成。

 说明：CLS 定义了编译器的输出需要，而编译器必须以 CLR 为编译宗旨。

1.3 Visual Studio 2010 有哪些新增功能

针对不同的使用者，Visual Studio 2010 共推出 5 个版本，分别是专业版、高级版、旗舰版、学习版和测试版。

- ❑ 专业版（Professional）：面向个人开发人员，提供集成开发环境、开发平台支持、测试工具等。
- ❑ 高级版（Premium）：创建可扩展、高质量程序的完整工具包，相比专业版增加了数据库开发、Team Foundation Server（TFS）、调试与诊断、MSDN 订阅、程序生命周期管理（ALM）。
- ❑ 旗舰版（Ultimate）：面向开发团队的综合性 ALM 工具，相比高级版增加了架构与建模、实验室管理等。
- ❑ 测试专业版（Test Professional）：简化测试规划与人工测试执行的特殊版本，包

含 TFS、ALM、MSDN 订阅、实验室管理、测试工具。

- ❑ 学习版 (Express)：是一个免费工具。提供了新的集成开发环境，适合非专业开发人员、学生等用户。

在 Visual Studio 2010 中，微软采用了全新的 WPF 技术重新打造了它的编辑器，新的编辑器，以及 Visual F# 获得了更加强大的功能，成为更好的 Web 开发工具。例如支持代码窗口的无级缩放、多窗口即时更新和代码的自动产生等，这些新的 IDE 特性都会极大地提高程序员的开发效率。

1. 更好的用户体验

在 Visual Studio 2010 中，明了的 UI 设计、增强的编辑器、对浮动文档和窗口的更好支持、增强的文档 targeting，以及行为回馈的焦点动画等功能提供了更好的用户体验，令流程更加自然，更加便于理解。

2. 更方便的Web开发

Visual Studio 2010 提供了高性能以及标准化的 JavaScript IntelliSense 引擎、“一键部署”能够快速将文档和设置发布到将要部署的网站上、对 Silverlight 的全面支持等功能增强，使 Web 开发工具的功能得到提升。

3. 对云计算的支持

Visual Studio 2010 包含了 Windows Azure 工具，可以简单的实现在微软云平台上的开发、调试，以及部署。提供了 C#，以及 VB 云服务项目的模板、更改 Service Role 设置的工具、本地开发整合 Development Fabric 以及 Development Storage 服务、在 Development Fabric 下的对 Cloud Service Roles 的调试、建立云服务软件包、浏览 Azure Services Developer Portal 等云计算的相关功能。

4. 更多数据库支持

现在，开发者除了 SQL Server，还可以在 IBM DB2，以及 Oracle 数据库下工作。IBM 大力提供了一个 Database Schema Provider (DSP)，这个 DSP 可以让 DB2 在 Windows、Linux 或 Unix 平台上工作。在 VSTS 2010 开发版中，它可以实现离线设计、开发、测试，以及更改管理等功能。

5. 并行编程

在 Visual Studio 2010 中，并行编程被简化，本地代码和管理代码的开发者都能够建立具有创造力的应用。

当然，Visual Studio 2010 离不开 .NET Framework，在 Visual Studio 2010 中支持的 .NET Framework 版本包括 2.0、3.0、3.5 和新增加的 4.0。.NET Framework 4.0 提供了以下一些新功能和改进特征。

- ❑ 应用程序兼容性和部署：除了一些在安全、标准遵从、正确性、可靠性及性能等方面的改进之外，.NET 框架 4 与基于早期 .NET 框架版本构建的应用程序高度兼容。

- ❑ 内核新功能及改进：对诊断和性能、垃圾收集、代码契约、设计时互操作程序集、动态语言运行时、协变与反变、元组、内存映射、64 位操作系统与进程等方面都进行了改进。
- ❑ 托管扩展框架：这是 .NET Framework 4.0 中的一个新库，帮助用户构建可扩展的和可组合式应用程序。
- ❑ 并行计算：针对编写多线程和异步代码引入了一个新的编程模式，从而极大地简化了应用程序和库开发者的编程。
- ❑ 网络编程：包括身份验证、支持 IPv6 等方面都得到了提升。
- ❑ Web 开发：对核心服务、Web 窗体、Web 窗体控件、动态数据、MVC 框架等各方面都引入了一些新特点。
- ❑ 客户端开发：WPF 在许多方面都发生了变化并进行了改进，包括控件、图形和 XAML 等。

1.4 Visual Studio 2010 开发环境介绍

经过前两节的介绍，相信读者对 Visual Studio 2010 已经有了一定的认识。在本节中，笔者将带领读者真正近距离地接触 Visual Studio 2010，看看其庐山真面目，并将初步给读者演示一下 Visual Studio 2010 的基本功能。

1.4.1 安装 Visual Studio 2010

在介绍 Visual Studio 2010 的安装步骤之前，请大家先确定一下自己的计算机操作系统的类别及版本。Visual Studio 2010 支持 Windows XP SP3 / Windows Vista SP1/Win 7/Windows Server 2003 SP2/Windows Server 2008 SP2 以上版本。表 1.1 具体列出了其运行环境的基本配置。

表 1.1 Visual Studio .NET 安装运行环境

Visual Studio .NET				
硬件设备	企业级结构设计版	企业级开发版	专业版	学院版
处理器	配有 1.6GHz 或更快处理器			
RAM	1024MB 以上			
可用硬盘空间	系统驱动器：600 MB、安装驱动器：3 GB（5400 转以上硬盘）			
DVD-ROM	需要			
鼠标	Microsoft 鼠标或兼容指点设备			
操作系统	Windows XP SP3、Windows Vista SP1、Windows 7、Windows Server 2003 SP2、Windows Server 2003 R2、Windows Server 2008 SP2、Windows Server 2008 R2			
视频	DirectX 9 视频卡，1280 x 1024 或更高显示分辨率			

初次安装 Visual Studio 2010 之前，基本不需要做预备组建的安装工作。但是，新版本

的软件对于操作系统的要求却更加严格。


 **说明：**建议在 Windows 7 以上版本上安装运行。

如果你已经是 Visual Studio.NET 系列的忠实用户，那么恭喜你，因为即使你的计算机上从来没有安装过 Visual Studio .NET 家族的任何产品，那么也不会安装在安装的第一步就出现“安装程序已检测到计算机上安装的某些系统组件与 Visual Studio.NET 要求的版本不匹配。你必须安装这些组件的另一个版本”这样的提示了。

其实，导致这个提示的直接原因是你的系统尚未安装 IIS 或者尚未设置 FrontPage 扩展服务器。Visual Studio.NET 默认的安装设置必须满足 IIS 组件的安装完成，并且完成了 FrontPage 扩展服务器设置的条件。找到操作系统安装盘，单击“开始”|“控制面板”|“添加或删除程序”命令，打开界面以后单击左侧的“添加/删除 Windows 组件”按钮，就可以安装 IIS 组件了。

FrontPage 扩展服务器设置视操作系统的不同，配置方法稍有差异。在 Windows 2003 操作系统中，用户可以在安装 IIS 的过程中选择配置 FrontPage 扩展服务器。其他操作系统可以在安装 IIS 之后，右击“IIS Web 站点属性”设置。

互联网信息服务（Internet Information Server，IIS），有的资料称为 Internet 信息服务，它实际上是一个万维网（World Wide Web）的 Server 服务。在 IIS 的支持下，用户可以在万维网中（WWW）发布网页，并且支持 ASP（Active Server Pages）、Java、VBScript 等产生的网络页面。

 **说明：**Visual Studio 2010 的内核已经取消了 IIS 及 FrontPage 扩展服务器设置的配备要求，如果只是在你的 PC 机上进行简单的程序开发与调试，那么就没有必要再次安装这些。但是，当你需要在服务器端发布这些网页或者服务程序的时候，还是需要单独安装 IIS 及 FrontPage 扩展服务器的。

为了便于本书后续章节的使用以及讲解，先向读者介绍一下 Visual Studio 2010 的完整安装过程。

如图 1.2 所示为 Visual Studio 2010 的安装界面。下面需要按照步骤，单击“安装 Microsoft Visual Studio 2010”按钮，进入安装必备组件的阶段。



图 1.2 Visual Studio 2010 安装程序步骤 1

进入 Visual Studio 2010 系统必备组件之后,就可以看到如图 1.3 所示的界面。这些都是安装 Visual Studio 2010 所必备的组件。Visual Studio 2003 之前的版本将这些信息放在了初始安装文件之后,那样会造成初级用户无法找到安装必需文件而引发的安装失败。Visual Studio 2005 以后的版本均将这些必备组件的安装文件直接置于整体安装之前,这样便省略了用户的手动寻找与安装的步骤,提高了软件本身的通用性。

单击“继续”按钮以后,进入必备组件的安装步骤中。按照提示完成必要信息的填写之后,可以单击“下一步”按钮完成整个 Visual Studio 2010 产品的安装。

在安装过程中可以根据自己的实际需要,安装相应的功能组件。Visual Studio 2010 为用户提供了包括 4 种语言在内的不同的开发工具。在 .NET Framework 平台下用户可以根据自己的需要选择安装语言种类及组件。完成以后,Visual Studio 2010 安装所必备的组件工具基本上已经全部安装在你的计算机上了。这时,会再次返回到安装主界面,进入 MSDN 的安装过程中。一般而言,如果是光盘安装,会有一个特定的文件夹保存着所有当前版本的 MSDN 信息。在输入验证码安装完毕后,用户可以通过链接直接在 Microsoft 的网站注册,这样就可以随时定期获得 MSDN 的免费支持了。当然,如果用户不满足于免费支持的资料及服务,也可以付费成为高级会员。这部分内容将在后续章节讨论。



请退出所有应用程序,然后再继续安装。

① 安装程序检测到已安装了以下所需的组件:

- Microsoft 应用程序错误报告

② 安装程序将安装下列组件:

- VC 9.0 Runtime (x86)
- VC 10.0 Runtime (x86)
- VC 10.0 Runtime (x64)
- Microsoft .NET Framework 4
- Microsoft Visual Studio 2010 64 位系统必备 (x64)
- Microsoft Visual Studio 2010 旗舰版


您必须接受这些许可条款,并在提示时输入 25 个字符的有效产品密钥才能进行安装。

图 1.3 系统必备安装界面

1.4.2 Visual Studio 2010 提供哪些项目模板

安装成功后,桌面上就生成了一个 Visual Studio 2010 快捷图标,可以打开软件认识一下 Visual Studio 2010 的真面目。

Visual Studio 2010 欢迎界面的布局基本沿袭了 Visual Studio .NET 的整体风格,与 2005 或者 2008 的差别不大。当用户进入欢迎界面以后,可以按照向导完成 .NET 平台下的初次配置。

 **注意:** Visual Studio 2010 提供了 3 种新建目标的类型,分别是新建项目、新建网站,以及新建文件。单击菜单栏的“文件”|“新建”命令就可以看到这些新建选项。

当要完成一个整体配置软件系统的时候,“项目”是一个非常不错的选择。如图 1.4 所示,在 Visual C# 项目中,不仅可以完成 Windows 应用程序的构建,而且可以完成 Windows 控件库、Windows 服务、类库等的建设。这些都是一个完备的大型项目所必需的要素。正是由于这些类型文件的必备元素已经被封装在各种模板当中,因此才能具体实现 .NET 的高效及完备的面向对象性。

为了便于读者了解并掌握各个模板的用途,表 1.2 诠释了各个主要模板的主要功能。



图 1.4 新建项目

表 1.2 项目模板

项目类型		Visual Studio 模板	功 能
Visual C#	Windows	Windows 应用程序	主要用于创建具有 Windows 用户界面的应用程序
		Windows 控件库	用于创建在 Windows 应用程序中使用的控件
		控制台应用程序	用于创建命令行应用程序
		空项目	用于创建本地应用程序的空项目
		类库	用于创建 C#类库 (.dll)
		Web 控件库	用于创建在 Web 应用程序中使用的控件
		Windows 服务	用于创建 Windows 服务
	Office	Crystal Reports 应用程序	用于创建 C#具有 Windows 用户界面以及一个 Crystal Report 示例的应用程序
		Excel 工作簿	用于创建基于新建的或现有的 Excel 2007 或 Excel 2010 工作簿来创建托管代码扩展的项目
		Excel 模板	用于基于新建的或现有的 Excel 2007 或 Excel 2010 模板来创建托管代码扩展
其他项目类型	安装和部署	Outlook 外接程序	用于创建为 Outlook 2007 或 Outlook 2010 托管代码外接程序的项目
		Word 文档	用于基于新建的或现有的 Word 2007 或 Word 2010 文档来创建托管代码扩展
		Word 模板	用于基于新建的或现有的 Word 2007 或 Word 2010 模板来创建托管代码扩展
	安装和部署	安装项目	用于创建 Windows Installer 文件, 可以实现向其中添加文件的项目
		Web 安装项目	用于创建可向其中添加文件的 Windows Installer Web 项目

续表

项 目 类 型		Visual Studio 模板	功 能
其他项目类型	安装和部署	合并模块项目	用于创建可向其中添加文件的 Windows Installer 合并模块项目
		安装向导	用于实现借助于向导的帮助来创建 Windows Installer 项目
		CAB 项目	用于创建可向其中添加文件的 CAB 项目
	扩展性	Visual Studio 外接程序	用于创建可以基于 Visual Studio 的主机上加载的外接程序
		共享的外接程序	用于创建可以在很多主机上加载的外接程序
数据库	SQL Server		用于创建一个允许直接操作数据库对象和数据的新数据库项目

1.4.3 Visual Studio 2010 提供哪些网站模板

假如用户只是希望自己能够完成一段网络程序的开发,例如 ASP.NET 程序,那么使用“新建项目”就显得不合时宜了。此时,Visual Studio 2010 可提供“新建网站”来支持所有网络项目的建设。表 1.3 给出了新建网站的主要模板功能。

表 1.3 网站模板

Visual Studio 模板	功 能
ASP.NET 网站	用于创建一个空的 ASP.NET 网站的模板
ASP.NET Web 服务	用于创建 XML Web Services 的网站
ASP.NET 网站(Razor)	使用 Razor 语法创建网站的项目
ASP.NET 空网站	用于创建一个没有模板的空网站项目
ASP.NET Crystal Reports 网站	用于创建一个带有 Crystal Report 示例的 ASP.NET Web 站点综合项目

1.4.4 Visual Studio 2010 提供哪些文件模板

当用户还没有想好要建立一个什么样规模的项目或者文件时,就可以使用“新建文件”。在之后的一段时间内,读者将会常常看到对这项功能的使用。表 1.4 给出了新建文件的主要模板功能。


说明:“新建文件”也可以被用在某种语言片断程序编译或者原型建立的阶段,也可以被用来新建用户自己的模板库。

表 1.4 文件模板

Visual Studio 模板	功 能
Web 窗体	用于创建 Web 应用程序的窗体
母版页	用于创建 Web 应用程序的母版页
Web 用户控件	用于创建使用可视化设计器的 ASP.NET 服务器控件

续表


Visual Studio 模板	功 能
HTML 页	用于创建可包含客户端代码的 HTML 页
Web 服务	用于创建 Web 服务的可视设计类
类	用于创建空类声明
样式表	用于创建 Rich HTML 样式定义的级联样式表
全局应用程序类	用于创建处理 Web 应用程序事件的类
Web 配置文件	用于创建配置 Web 应用程序设置的文件
XML 文件	用于创建单独的空白的 XML 文件
XML 架构	用于创建 XML 文档架构的文件
文本文件	用于创建空白文本文件
资源文件	用于创建 .NET 资源文件
SQL 数据库	用于创建空 SQL 数据库
数据集	用于创建包含 DataSet 类的 XML 架构的文件
一般处理程序	用于创建用于实现一般处理程序的页
站点地图	用于创建站点地图的文件
Crystal 报表	用于创建向 Windows 或 Web 窗体发布数据的 Crystal 报表文件
VBScript 文件	用于创建包含 VBScript 代码的脚本文件
移动 Web 窗体	用于创建适于移动 Web 应用程序的窗体
报表	用于通过 Microsoft 制表技术创建报表的空白报表文件
JScript 文件	用于创建包含 JScript 代码的脚本文件
移动 Web 用户控件	用于创建适合在移动 Web 窗体上使用的 ASP.NET 服务器控件
XSLT 文件	用于创建用于转换 XML 文档的文件
移动 Web 配置文件	用于创建用于配置移动 Web 应用程序设置的文件
外观文件	用于定义 ASP.NET 主题的文件
浏览器文件	用于定义浏览器定义的文件
类关系图	用于创建空白的类关系图

根据实际需求合理地创建适当的类型项目,是 Visual Studio.NET 一直推行的开发理念。希望读者能够在接下来的学习与操作过程中,深入领会这种“分门别类”的程序开发模式。

1.4.5 与 Visual Studio 2010 的第一次相会

IDE (Integrated Develop Environment), 即集成开发环境,它集成了代码编写、分析、编译、debug 等功能于一体,包括代码编辑器、编译器、调试器和图形用户界面工具。Microsoft 的 Visual Studio 2010 是指包含了所有 IDE 必备特性的软件组。

对于一个开发工具来讲,集成开发环境的好坏,直接决定着开发人员在实际的软件开发过程中能否进行高效率的工作。当用户每次启动 Visual Studio 的时候,集成开发环境中会出现一个默认的主页,通过单击这个主页所提供的链接,用户可以快速打开曾经建立的项目或者直接新建项目。

 **说明：**除了对项目文件的选择之外，用户也可以通过页面上提供的链接进入联机帮助文档或进行信息搜索，并且还可以对集成开发环境的配置进行修改等。

在默认主页中选择“新建项目”命令，给这个项目起名为 Example1。在对话框中配置完成后，单击“确定”按钮，就进入 Visual Studio 2010 的开发界面了，这时就可以看到如图 1.5 所示的操作界面的布局。

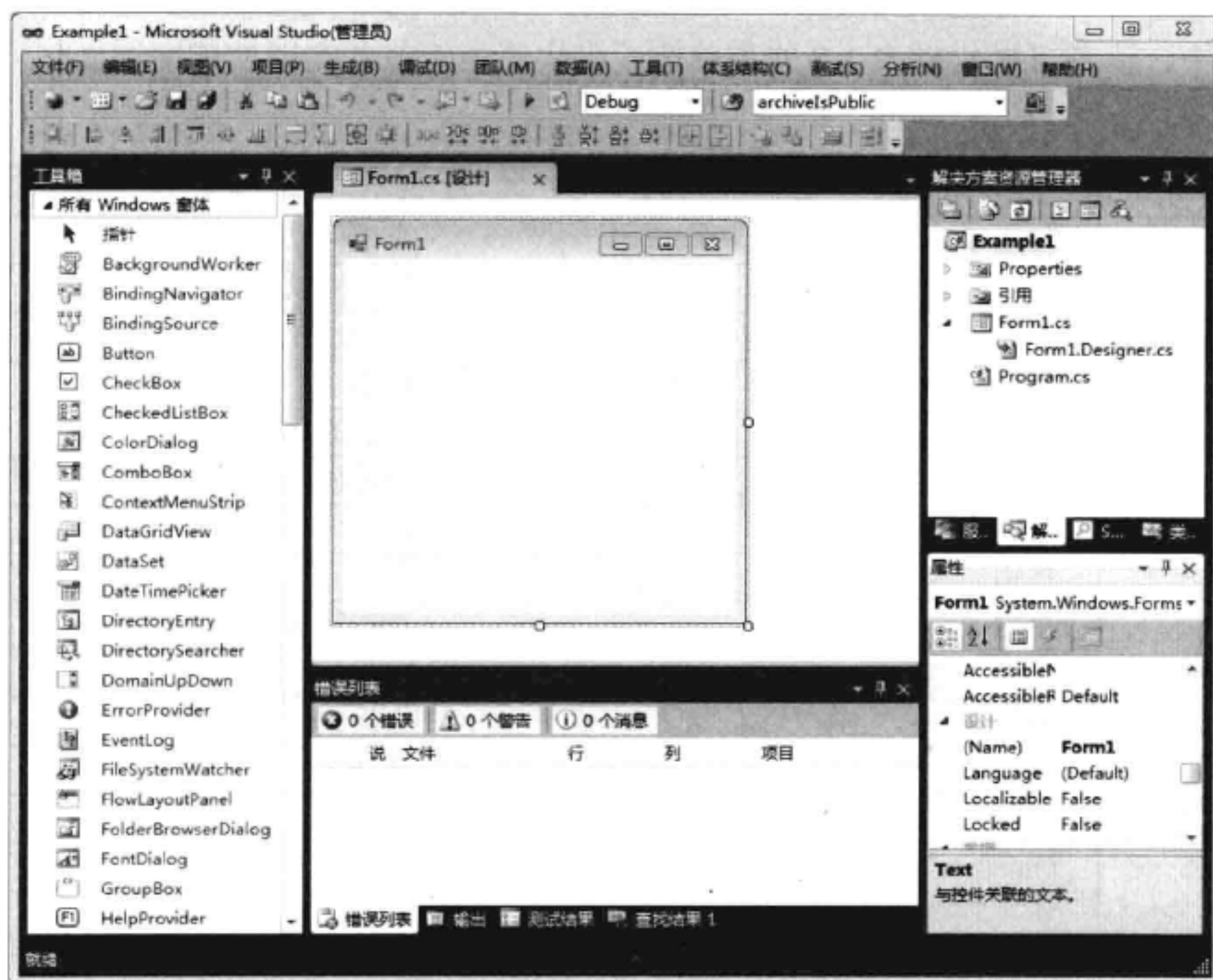


图 1.5 Visual Studio 2010 IDE

Visual Studio 2010 主要包括如下几个方面：

- ☐ 菜单栏；
- ☐ 开发工具栏；
- ☐ 解决方案资源管理器；
- ☐ 工具箱；
- ☐ 窗口管理；
- ☐ 类视图；
- ☐ 属性窗口；
- ☐ 资源视图。

在接下来的各小节中，将分别介绍 IDE 中主要的操作项的功能及设置方法。

1.4.6 必须熟悉的开发工具栏和菜单栏

进入项目开发环境后，首先被用户所看到的的就是菜单栏以及开发工具栏。相信读者都

不会对 Windows 操作系统下的菜单栏感到陌生，在 Visual Studio 2010 中，菜单栏是统领全局的关键所在。菜单栏如图 1.6 所示。

文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) 团队(M) 数据(A) 工具(T) 体系结构(C) 测试(S) 分析(N) 窗口(W) 帮助(H)

图 1.6 菜单栏

关于菜单栏的具体选项页设置，可以通过表 1.5 做一番清楚的解释。在本书接下来的操作讲解中，读者可以根据表 1.5 的菜单选项来快速地查找并实现功能配置操作。

表 1.5 菜单栏选项列表

菜 单 项	子 菜 单 项	
文件	新建项目/网站/文件	全部保存
	打开项目/网站/文件	导出模板
	添加	页面设置
	关闭	打印
	关闭项目	最近的文件
	保存	最近使用的项目和解决方案
	另存为	退出
编辑	撤销	全选
	重做	查找和替换
	剪切	转到
	复制	定位到
	粘贴	书签
	删除	
视图	Quick Access	错误列表
	解决方案资源管理器	输出
	Solution Navigator	起始页
	团队资源管理器	任务列表
	服务器资源管理器	工具箱
	体系结构资源管理器	查找结果
	书签窗口	其他窗口
	调用层次结构	工具栏
	类视图	全屏显示
	代码定义窗口	Tab 键顺序
	文档大纲	属性窗口
	对象浏览器	属性页
项目	添加 Windows 窗体	从项目中排除
	添加用户控件	显示所有文件
	添加组件	添加引用
	添加类	添加服务引用
	添加新项	管理 NuGet 程序包
	添加现有项	启用 NuGet 程序包还原
	添加新的分布式系统关系图	属性
生成	生成解决方案	清理文件
	重新生成解决方案	发布文件

续表

菜单项	子菜单项	
生成	清理解决方案	对文件运行代码分析
	生成文件	批生成
	重新生成文件	配置管理器
调试	窗口	新建断点
	启动调试	删除所有断点
	开始执行（不调试）	IntelliTrace
	附加到进程	清除所有数据提示
	异常	导出数据提示
	逐语句	导入数据提示
	逐过程	选项和设置
	切换断点	
数据	显示数据源	数据比较
	添加新数据源	Transact-SQL 编辑器
	架构比较	架构视图
格式	顺序	垂直间距
	对齐	在窗体中居中
	使大小相同	锁定控件
	水平间距	
工具	附加到进程	扩展管理器
	连接到数据库	Dotfuscator Software Services
	连接到服务器	WCF 服务配置编辑器
	代码段管理器	外部工具
	选择工具箱项	导入和导出设置
	外接程序管理器	自定义
	库程序包管理器	选项
	宏	
体系结构	新建关系图	窗口
	生成依赖关系图	
测试	新建测试	管理测试控制器
	加载元数据文件	选择活动测试运行配置
	创建新测试列表	编辑测试设置
	运行	窗口
	调试	
分析	启动性能向导	对项目配置代码分析
	启动 HPC 性能向导	为解决方案配置代码分析
	比较性能报告	为所选项目计算代码度量值
	探查器	为解决方案计算代码度量值
	对项目运行代码分析	窗口
窗口	新建窗口	Pin Tab
	拆分	自动全部隐藏
	浮动	新建水平选项卡组
	停靠	新建垂直选项卡组
	以选项卡式文档停靠	关闭所有文档
	自动隐藏	重置窗口布局

续表

菜 单 项	子 菜 单 项	
窗口	隐藏	窗口...
帮助	查看帮助	索引结果
	管理帮助设置	注册产品
	MSDN 论坛	检查更新
	报告 Bug	技术支持
	示例	联机隐私声明
	客户反馈选项	关于 Microsoft Visual Studio

每一系列工具栏的尾部都会有一个 ▼ 形状的下拉按钮，用鼠标单击这个按钮并连续进入子菜单，最终可以对工具栏上的工具按钮进行定制。在子菜单上，用户可以通过复选对应的项目将工具按钮放置到当前的工具栏上，也可以将某个工具按钮从当前工具栏上去掉。

选择“视图”|“工具栏”命令或者选择“工具”|“自定义”命令会出现一个对话框，在标签页选项可以选择工具栏的选项，如图 1.7 所示。



图 1.7 工具栏选项标签

关于工具栏的定制方法以及作用，将在 1.5 节中重点讨论。

1.4.7 用对象浏览器查看对象信息

对象浏览器可以让用户查看包括名称空间、类、结构、接口、类型和枚举等类型的各种对象的信息，以及它们的成员信息。这些成员信息包括属性、方法、事件、变量、常量和枚举条目等。所有这些对象可以包含在当前项目中，也可以包含在当前项目引用的组件中。

选择“视图”|“对象浏览器”命令，窗口管理标签页上就出现了如图 1.8 所示的界面。打开 Example1 节点，可以看到在对象浏览器下的项目 Example1 的所有对象信息，如图 1.9 所示。



图 1.8 对象浏览器

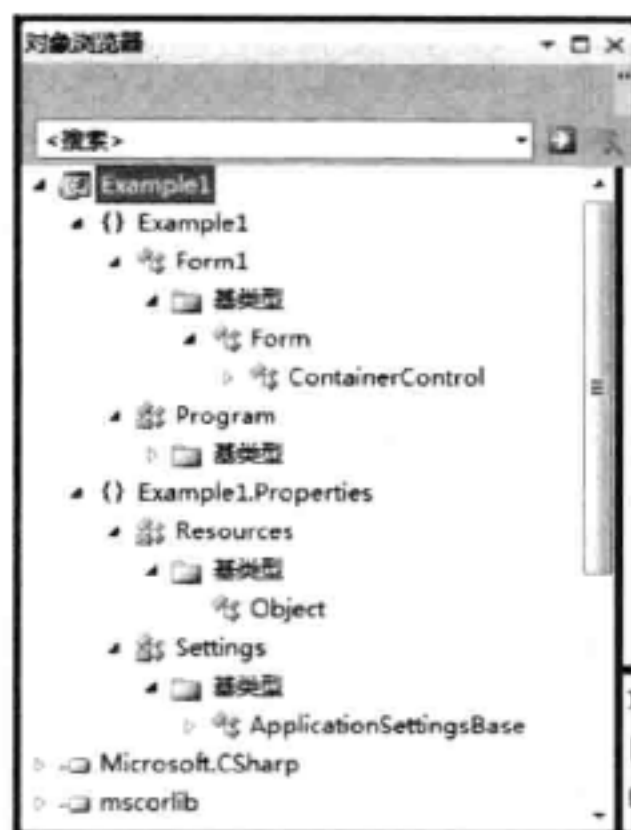


图 1.9 示例对象浏览器

除了以“解决方案”为根节点的树形的项目视图之外，IDE 还为用户提供了面向对象编程所必需的类视图。

说明： Visual Studio 2010 提供了几种不同的视图形式。

选择“视图”|“类视图”命令，出现如图 1.10 所示的界面。在类视图窗口中展开 Example1 节点，可以看到用图标显示的节点包含了项目引用、对该类中存在的信息映射函数的反映和接口信息，如果用鼠标双击对应项目，则立即转到代码的对应位置。

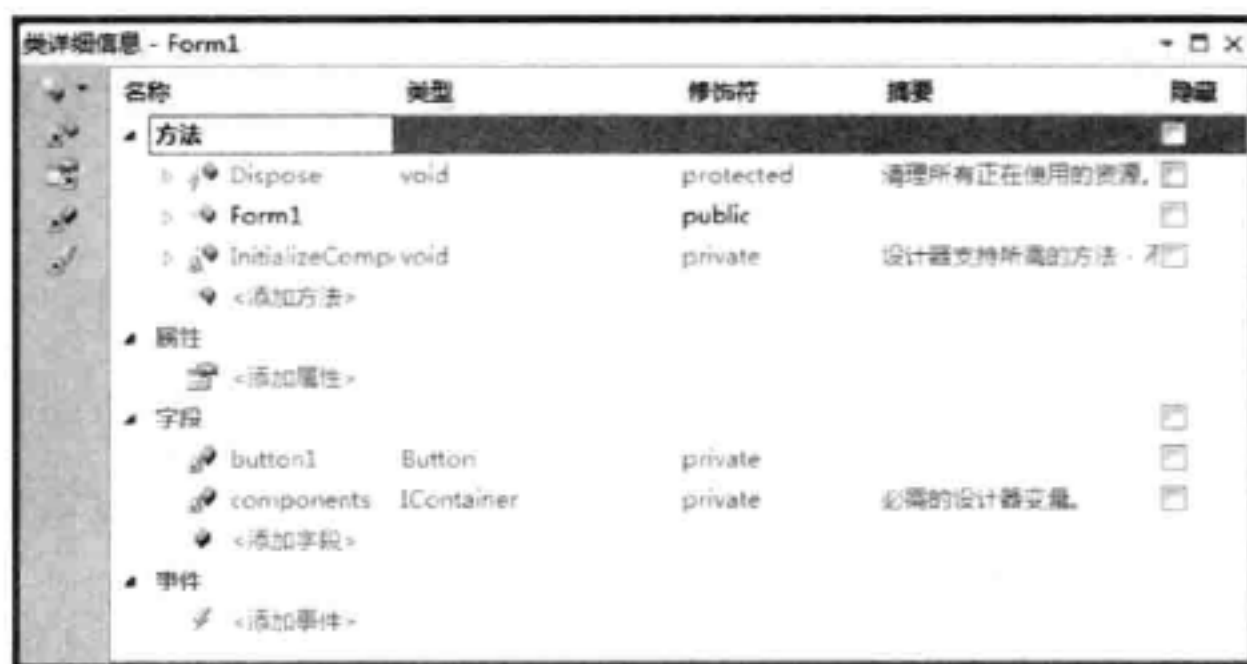
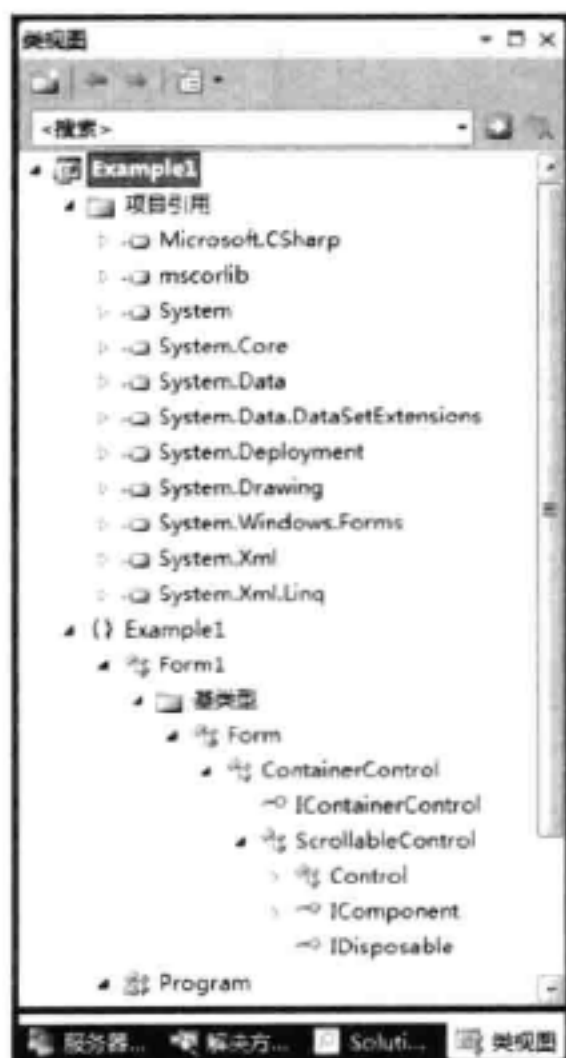


图 1.10 类视图

1.4.8 可视化利器：工具箱和属性窗口

工具箱为用户提供了许多新的用于 Web 窗体、Windows 窗体，以及数据设计的控件，

如图 1.11 所示。这些控件极大地提高了开发人员的工作效率。

一般情况下，工具箱在项目开发界面的左侧隐藏。用户只要将鼠标移至左侧边界靠近工具箱图标，它就会自动出现。如果用户需要固定工具箱，那么仅需要单击工具箱右上角的楔形图标即可。这样，工具箱便被固定在 IDE 的可视界面上。

如果用户想要将工具箱移至界面的其他地方，或者单独置于页面整体平面之上，可以将鼠标移动至工具箱标题的蓝色空白处，然后选择“拖放”命令即可。其他标签页及工具栏都可以同理操作。

属性窗口一般隐藏于界面右侧，如图 1.12 所示，用户可以像工具箱操作一样，将属性窗口设置为可见。当然，也可以选择“视图”|“属性窗口”使其命令可见。

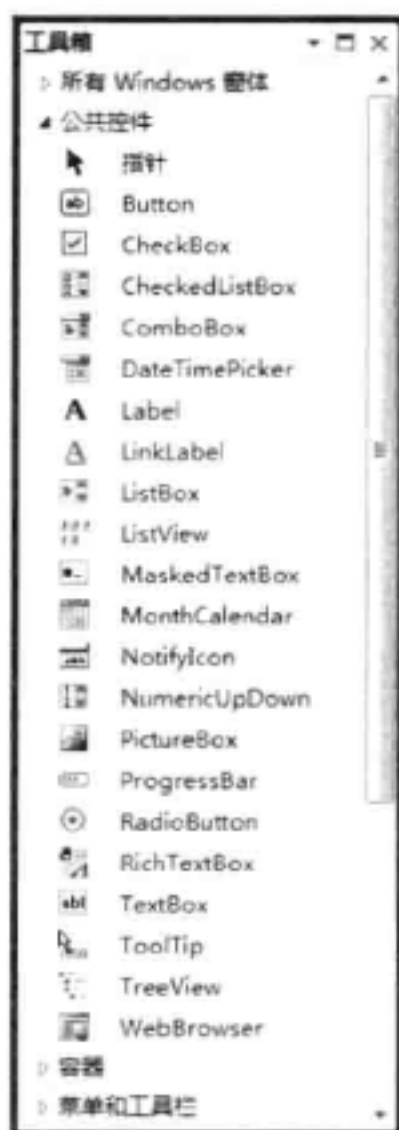


图 1.11 工具箱面板



图 1.12 Windows 窗体属性窗口

使用属性窗口可以让用户在设计期间查看和修改在编辑器或设计器中选定对象的属性和事件。

注意：用户通过属性窗口还可查看编辑文件、项目，以及方案的属性。

在 Form1 的属性窗口中，能看到与此窗体相关的属性列表，这些属性值可以帮助用户进行最基本的 Windows 窗体界面设计。在以后的学习过程中，将逐一介绍不同控件的属性值及其作用。

属性选项框中，在框体最上方的文本框内显示的是当前操作控件的名称，以及所属类。现在新建的 Form1 就是 System.Windows.Forms.Form 类的一个实例。这不仅体现了 C# 语言的面向对象性，还体现了其类型的统一性。这些将在后面章节进行讨论。

在属性选项框中文本框的下面是一排可选择的按钮。它们代表的操作命令分别是“按分类排序”、“字母排序”、“属性”、“事件”还有“属性页”。单击“事件”标签后，可以看到 Windows 窗体所能引发的操作事件列表。

在了解了 Visual Studio 2010 的组成和各部分的基本功能之后，在 1.5 节将重点介绍集成开发环境中各种环境属性的设置方法。

1.5 定制环境

在 Visual Studio 2010 中可以改变很多属性。本节以 1.4.7 节中建立的项目 Example1 为例，在这个实例中为读者具体演示一下 IDE 环境的设置与更改。

1.5.1 让字体和颜色更适合自己

有过一定开发经验的人知道，工具栏可以用来简化常用操作。但是，界面上如果存在过多的工具栏或一个工具栏上集中过多的工具按钮，就会占用代码编辑窗体的空间。这样不但不会给用户的工作带来方便，反而会影响工作效率。为了解决这个问题，集成开发环境为工具栏提供了更为方便的定制功能。

选择“工具”|“选项”命令，弹出选项对话框。单击“环境”标签，选择“字体和颜色”子标签。在如图 1.13 所示的界面中，可以完成对 IDE 字体，以及颜色的控制。

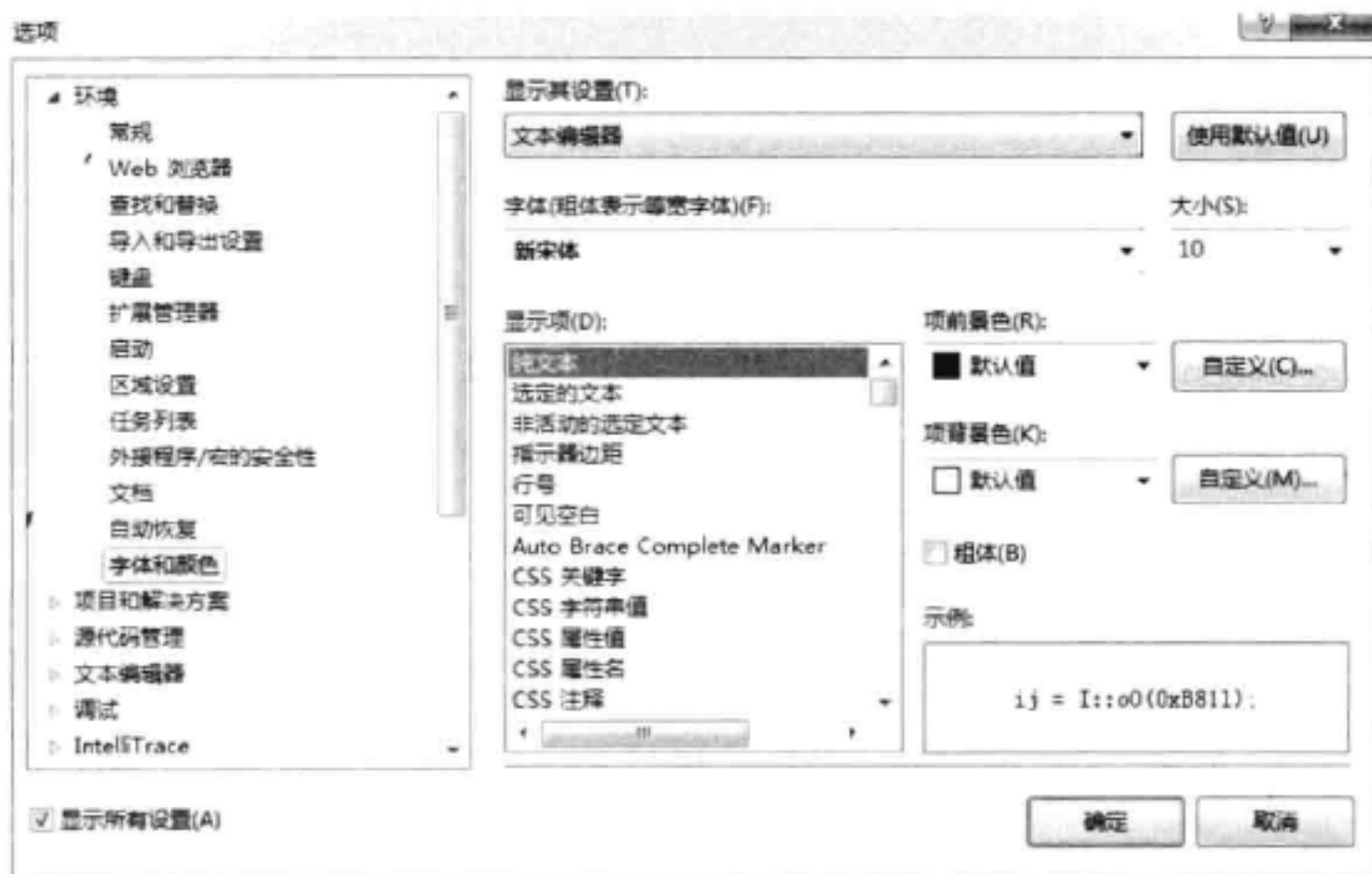


图 1.13 修改字体和颜色

在这个对话框的标签中，不仅可以根据提示修改字体和颜色，更可以设置这个解决方案的许多属性，用户可以根据实际需要来完成操作。

1.5.2 在项目和解决方案中保存文件

在一个项目中，添加任意多个文件，这些文件可以是不同类型、不同版本的。只要在 .NET Framework 平台上，它们都能够很好地通过接口文件有效地整合在一起。下面就来简单介绍一下如何在项目和解决方案中保存文件。

需要保存当前项目文件中的某个文件时，可以在 IDE 中打开该文件，然后选择“文件”|“保存”命令，在 IDE 界面中的 Form1 就被保存了下来。在实际编程过程中，也可以利

用快捷键 Ctrl+S 来保存文件。

如果需要在—个解决方案下添加多个项目文件，或者在项目下添加多个子文件的时候，可以右键单击需要添加子文件的项目文件，这时候会弹出一个菜单选项，选择“添加”按钮，就会出现可以添加在该项目或解决文件下的文件类型，如图 1.14 和图 1.15 所示。

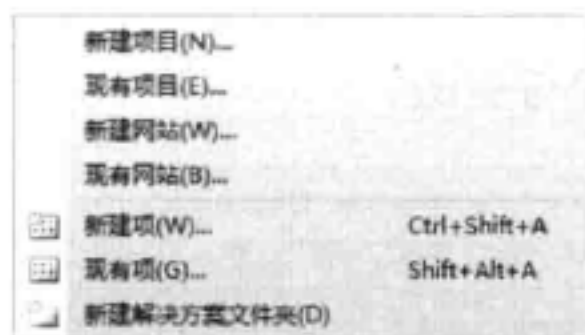


图 1.14 解决文件添加项

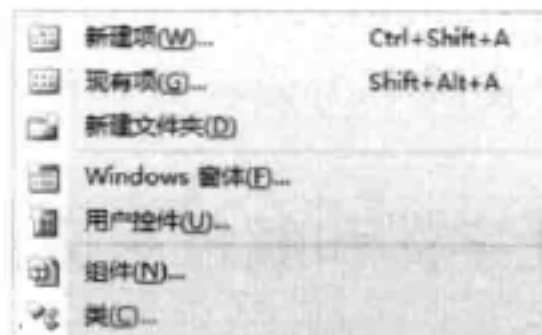


图 1.15 项目文件添加项

选择需要添加的文件类型以后，就可以按照新建—个文件的方法，在已有的解决方案或者项目文件下成功添加—个新的文件。

技巧：这个文件一旦被添加在某个节点下，那么它在这个项目中也就不存在了。用户只需要在操作完之后，以同样的步骤单击“保存”按钮就可以了。

1.5.3 使用任务列表和注释管理代码

为了对代码进行组织和管理，用户往往需要对自己编写的代码进行详细的注释，以方便自己和其他人来理解代码含义。以往，这个功能一般依赖于对代码进行常规注释的方法来达到目的，但是常规方法在多个文件中作标记时，无法让用户快速地找到他们所关心的位置。

为了更好地对这些提示性的注释进行良好的管理，Visual Studio 2010 集成开发环境中引入了任务列表功能。这个功能可以使用户在—个窗口中，对包含标记为 TODO、HACK 和 UNDONE 等注释关键字的位置进行保存和显示。而且，用户也可以对关键字进行自定义。自定义的步骤为选择“工具”|“选项”命令进入提示框，在“环境”|“任务列表”中添加即可。从任务列表这个窗口中，对选择项目用鼠标双击的方式直接切换到代码的对应位置。这个功能充分满足了开发人员对代码助记功能的需求。任务列表窗口如图 1.16 所示，在代码的下方。在 Form1 中添加了一段用 TODO 关键字标识的注释 Todo this is an explanation，而任务列表中显示出了这段代码的内容以及行号等信息。

在 Visual Studio 2010 环境中，所有语言的注释都是以“//”来规范书写的。在 Form1 的界面上，右键单击出现提示菜单，选择“查看代码”命令，就进入到代码展示的界面。在代码当中，能够看到用“//”标出的代码注释。


1.5.4 在命令窗口中执行命令

之前在学习 Visual Studio IDE 的菜单栏时，读者可能就发现了有些命令处于二级菜单中，操作起来十分不方便。Visual Studio 2010 为程序员提供了直接的命令窗口，开发人员

可以直接输入命令语句，以命令行的形式来执行各个二级菜单中的操作命令。



图 1.16 任务列表与注释

 **说明：**开发人员也可以直接输入一级菜单中的命令项，这样就实现了用键盘完全代替鼠标的工作。

选择“视图”|“其他窗口”|“命令窗口”命令，就会出现图 1.17 所示命令窗口。

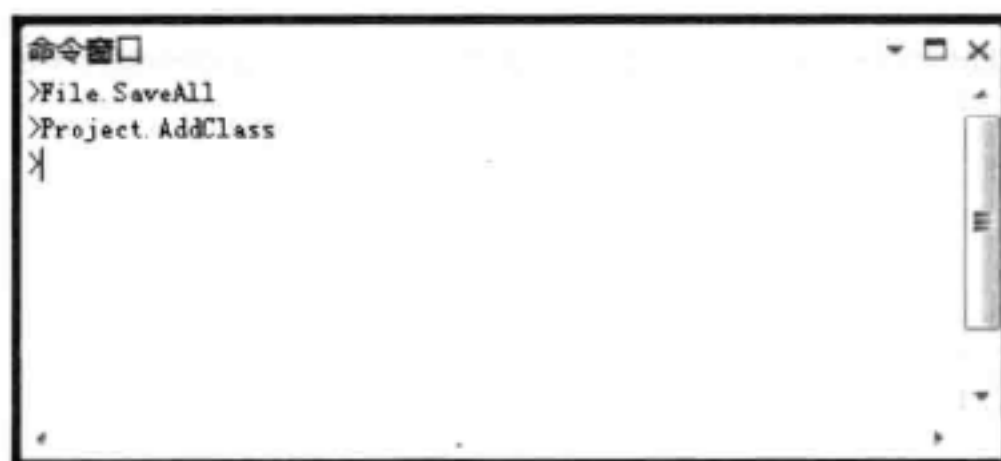



图 1.17 命令窗口

在“>”符号后面，程序员可以通过输入相应英文菜单项名称来实现鼠标操作。如果是二级菜单项中的命令，那么在一级菜单项命令名称之后加入“.”即可。这有点类似于对类中方法函数的调用。如图 1.17 所示，在“>”提示符后输入 Project.AddClass，然后按 Enter 键就可以实现选择“项目”|“添加新类”命令的操作，就会弹出相应的对话框。输入命令行以后，可以通过单击菜单来验证这项操作的一致性。

通过观察，相信大家已经发现，命令行使用的都是英文命令符。其实，这些英文命令符就是各级菜单的英文选项名称。如果安装的是英文版的 Visual Studio 2010，那么掌握起来可能会有事半功倍的效果。

 **说明：**命令窗口为开发人员提供了一个快捷的窗口，使程序开发人员可以在传统菜单和工具栏基础上能够自主选择应用自己喜爱的操作方式。

1.5.5 代码显示行号与代码折叠

Visual Studio 2010 IDE 最大限度地实现了对程序开发者的支持，细微之处令所有的程序员由衷为之叫好。相信有过程序开发经验的人都会感觉到，当程序的规模达到一定限度的时候，对代码的阅读就会无端地增加很多困难。例如，当开发人员忘记添加注释的时候，可能很难定位相关代码所在行。在开发团队中，人们利用行号来交流的时候并不一定会比注释符少。这种情况下，.NET 的相关支持会为所有的用户带来惊喜。

在之前建立的项目文件中，读者没有看到行号显示。现在，选择“工具”|“选项”命令，如图 1.18 所示。在对话框中选择“文本编辑器”|C#选项，在“显示”选项区域中选择“行号”复选框就可以改变行号的显示属性了。



图 1.18 行号显示属性

改变完毕后，代码已经有了行号显示。

在大型代码中，还可以自主查找固定行号所在行，如图 1.19 所示，双击 IDE 右下角的“行”显示，就会弹出如图 1.20 所示的对话框。利用这个对话框完全能够搜索到指定的代码行。



图 1.19 IDE 右下角展示区



图 1.20 转到行对话框

代码行的问题解决了，但是大型代码的行数“数量可观”的问题仍然摆在读者面前。此时就需要利用代码折叠功能了。如图 1.21 所示，在某些部位的程序代码左侧有一个符号标记，这个标记就是为了方便程序员折叠隐藏方法函数而设计的。

除了直接单击代码左侧的减号“-”以外，也可以选择“编辑”|“大纲显示”命令来决定操作。在这个标签下，可以选择其他显示代码的形式，如图 1.22 所示，这种操作在应

对繁杂代码的整体操作之时更为有效。

```

1  using ...
2
3  //todo This is an explanation
4
5  namespace Example1
6  {
7      public partial class Form1 : Form
8      {
9          public Form1()
10         {
11             private void Form1_Load(object sender, EventArgs e)
12             {
13             }
14         }
15     }
16 }

```

图 1.21 代码折叠

切换大纲显示展开(T)	Ctrl+M, Ctrl+M
切换所有大纲显示(L)	Ctrl+M, Ctrl+L
停止大纲显示(P)	Ctrl+M, Ctrl+P
停止隐藏当前区域(I)	Ctrl+M, Ctrl+U
折叠到定义(O)	Ctrl+M, Ctrl+O

图 1.22 大纲显示标签

折叠后的代码左侧部分显示为一个加号“+”。当读者以后阅读冗长程序时，假如看到了这种符号，就可以确定其中藏有玄机了。

技巧：折叠后的代码更利于程序员统观全局，快速、及时地找到各个方法对象的位置所在。

1.5.6 管理 Visual Studio 2010 中的子窗口

现在，读者应该可以对 Visual Studio 2010 IDE 操作自如了。下面进入到定制环境所需要掌握的最后部分——窗口的操作。

窗体操作主要有以下特性：

- ☐ 分页标签；
- ☐ 自动隐藏；
- ☐ 任意停靠。

接下来分别介绍各个特性的实际体现。

1. 分页标签

在项目 Example1 中，添加了几个新的 Windows 窗体文件，分别命名为 Form2、Form3、Form4、Form5、Form6，然后分别打开项目中这 6 个窗体文件的代码文件，这时将会发现，这个 IDE 的显示栏变得混乱了。实际上，在大型项目中常会遇到同时打开多个文件的情况。程序员也经常需要在多个标签页之间来回切换。假如各个标签页距离太远，那无异于浪费时间与精力。所以，需要将一段时间内应用到的相关文件的标签移动到一起。这个操作其实很简单，只需要单击任何一个标签不放，然后按住鼠标左键直接将其拖动到标签行的某处就可以了，如图 1.23 所示。一个看似可有可无的操作，带来的效果却是惊人的。关于 Visual Studio 2010 IDE 的便捷之美，还需要读者在日后的学习过程中慢慢体会！

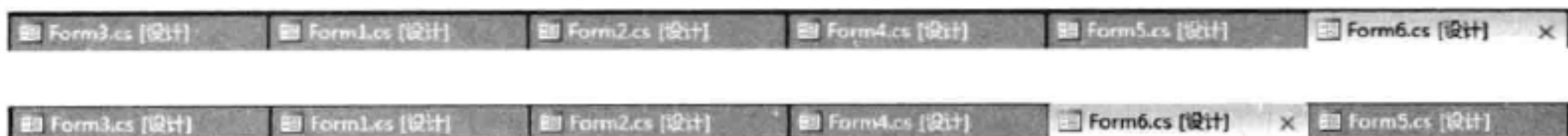




图 1.23 标签拖动比较

2. 自动隐藏


当 IDE 中的信息窗口打开过多时，整个 IDE 界面会显得混乱而无序。然而，所有的信息窗口又都是必须开启的，不能关闭。这种情况下该怎么办呢？Visual Studio 2010 IDE 为开发者提供了信息窗口自动隐藏的功能。最快捷的方法就是单击任意一个标题窗口右方的图钉按钮。这个按钮的图释可以表示已经开启或者关闭了自动隐藏功能。

此外，标题窗口的页眉处有一个按钮，单击这个按钮就可以选择此标题窗口的隐藏状态。如果认为寻找这个小小的按钮太麻烦，那么最迅速的更改状态的方式就是在任意标题窗口的标题栏单击鼠标右键，然后在“自动隐藏”标签处选择即可。

3. 任意停靠



所有的信息窗口都是停靠窗口，只要在某个信息窗口的标题栏单击鼠标左键并且按住不放，就可以拖动窗口了，直到窗口停留到所指定的位置以后，放开鼠标左键即可。开发人员可以根据自己的喜好安排停靠窗口的位置，进行任意排列。

当用户想要将已经悬浮的信息窗口再次停靠到之前最后一个停靠位置的时候，可以用鼠标双击标签窗口的页眉处，那么该窗口就会自动停靠在之前的位置了。同样，也可以在已经停靠的窗口页眉处再次双击鼠标，那样会令该窗口再次悬浮。

 **技巧：**用户也可以在任意窗口的页眉处右击鼠标，将“可停靠”标签前面的勾号去掉，那么该信息窗口就变为永远的悬浮状态了。

1.5.7 调试与生成程序

在了解了 Visual Studio 2010 IDE 的定制环境以后，本节最后要向读者讲述一些程序编写方面的常识。在利用 Visual Studio 2010 进行项目建设或者程序开发的过程中，必然也要逐步对编写的代码进行调试以及检验。这就得提及程序的调试与生成方面的知识了。

在 Visual Studio 2010 IDE 中，可以通过选择菜单栏的“生成”|“生成解决方案”命令（“重新生成解决方案”）的方法实现项目程序的生成。同理，选择“调试”|“启动调试”命令也可以完成调试工作。然而，在实际操作中，程序员一般选择在菜单栏空白处右击鼠标，选择“标准”以及“生成”标签来调用相应的快捷标签更加实际。之后，只要单击 IDE 中的按钮就可以完成调试工作，而单击按钮即可以完成生成操作。

有的读者可能要发问了：生成与调试有什么区别呢？生成仅是完成对代码的编译，即流程正确性检验，而不输出运行结果。而调试则是在生成成功之后输出运行结果。对应用程序而言，生成成功只是语义上的评判，没有什么实际的回馈；而调试成功则可以生成应用程序的界面或者服务，并且允许程序员设置断点完成程序跟进等。

总地看来，生成是调试基础，而调试必须建立在生成成功的基础上。生成过程包括如下内容：

- ☐ 将项目中的代码传输给编译器；
- ☐ 将源代码转化为 Microsoft 中间语言（MSIL）；
- ☐ 将 MSIL 和元数据与项目中的其他相关模块进行交互连接；

❑ 创建扩展名为 .exe 或 .dll 可执行文件。

在实际开发过程中，应当注意“生成”与“重新生成”的区别。当执行大型的项目时，“生成”只对刚刚修改过的文件进行重新编译，而“重新生成”则是对整个解决方案或整个项目进行全部的重新编译，因此程序员需要依据实际情况选择生成方式。仅仅运行“生成”比较高效，适用于局部高频次修改程序的情况。而在确立基线或最终发布程序时就必须用“重新生成”进行一次完全编译了，这样不仅可以确保整个项目的正确性及完整性，更加有利于版本控制。

可以在解决方案的标题处右击鼠标，则出现如图 1.24 所示的界面。



图 1.24 解决方案的快捷菜单

应用程序的调试版本和发布版本是不一样的，前者包含对调试器提供足够的调试信息，而后者则更多考虑运行时的优化。配置管理器的作用就是对两种主要的生成条件进行切换。选择“生成”|“配置管理器”命令，可以打开“配置管理器”对话框，如图 1.25 所示。



图 1.25 “配置管理器”对话框


在“配置管理器”对话框上方的“活动解决方案配置”下拉列表框中选择类型。在调试过程中一般要注意以下所述几点。

1. 多项目启动及单项目启动

在大型项目中，启动项一般包括前台的登录界面及后台的服务，所以需要多项目启动的支持。

2. Debug与Release的区别

简单而言，Debug 版本是调试版本，而 Release 版本则是发布版本。如果开发人员准备对应用程序进行调试，则应该在解决方案配置窗口选择 Debug 选项。假如需要生成正式版本，那么就通过下拉菜单选择 Release 选项即可。

说明：Debug 版本要比 Release 版本大得多。这主要是因为 Debug 版本包括了许多调试信息并且没有被优化的缘故。相对而言，Release 版本为了便于用户使用，所有的调试信息都已经被摒弃了，而且编译器自动调优到最大。


3. 配置

有些 XML 文件只需要在程序第一次 Debug 的时候生成一遍，以后的维护不需要，甚至禁止再次生成。所以在多项目的大型程序中，需要标出哪些项目需要启动配置。当然，有时这样做是为了提高编译的效率。

1.6 学会使用 MSDN 帮助系统

在安装完 Visual Studio 2010 重启计算机后，能够看到 MSDN 的安装项。很多用户觉得 MSDN 的作用并不是很大，安装后很占用系统的资源，因此往往不会主动安装 MSDN 文件。其实，这种做法是极大的损失。只有了解并掌握了 MSDN 帮助系统的使用方法，才能更好地理解 .NET 的各项功能，也才能将其功效发挥至最大。

MSDN 的全称是 Microsoft Developer Network，它是 Microsoft 面向软件开发者的一种信息服务。作为 .NET Framework 的配套环境，它是一个以 Visual Studio 和 Windows 平台为核心整合的开发虚拟社区。很多人误认为 MSDN 仅是联机帮助文件与技术文献的集合，这种观点是片面的。实际上，这两者只占 MSDN 庞大计划的一小部分而已。

说明：MSDN 不仅包括技术文档，还包括 Internet 在线电子教程、网络虚拟实验室、Microsoft 产品下载（几乎包括全部的操作系统、服务器程序、应用程序和开发程序的正式版和测试版，还包括各种驱动程序开发包和软件开发包）、MSDN WebCast、与 CMP 合作的 MSDN 杂志等一系列服务。

MSDN 的基本构成如表 1.6 所示，能够在 MSDN 中找到许多有价值的参考。

表 1.6 MSDN 产品列表


MSDN 产品	说 明
更新说明和更多的浏览信息	首页和 Welcome 信息
.NET 的文档	.NET 的规范、配置方法、API 说明、入门材料等，内容十分丰富
Visual Studio 的帮助库	很多 Visual Studio 的基本内容
Office 开发者文档	支持 Office 扩展开发，有很多 Office 教材和疑难解答
平台 SDK 开发文档	SDK 开发实例以及说明
Windows 系列资源包的开 发文档	Windows 系列下每个平台特性说明和独有特性 API 说明
嵌入设备开发者文档	将 eVC 和 eVB 这两个 MS 嵌入式开发工具的帮助结合到整个 MSDN 中来的，原有的文档都能在 MSDN 中找到
其他文档	新加入的 XML&SOAP 开发包、Passport 开发包、Project 2000 等，是变化最快的内容之一
技术文章	是 MSDN 网站上的问答集合，有很多有价值的信息
知识库	阐述了一些开发模式和性能优化的建议，分门别类列举了实现特定目标的一般模式和组织方法，可以作为大型项目的开发参考
背景知识	一些大的蓝图和规范说明，包括了很多方面，例如，组件对象模型、数据库和消息队列、Office 应用、Web 开发、Windows 系列平台的应用等
规格书	即白皮书，各种规范
有价值的书籍或文案资料	经典的书籍的节选
杂志节选	出自最近的 MSDN 等杂志
示例	全部例子的索引

通过 Internet 的连接，用户可以了解到 MSDN 最新信息、错误的报告、MS 最新的发展动态，以及更多的信息访问地址。

用户接触到的最多关于 MSDN 的信息可能便是来自于 MSDN Library 了。MSDN Library 就是通常所说的 MSDN，它涵盖了 Microsoft 全套可开发产品线的技术开发文档和科技文献，甚至包括部分的开发源代码。MSDN Library 每个季度更新一次，用户可以在 Microsoft 的官方网站上下载，下载之后直接安装就可以运行了，不用专门的连接。用户在使用的时候，在菜单栏的“帮助”选项里就可以找到它，十分方便。

用户还可以通过门户网站直接向 Microsoft 订阅更新光盘。由于 MSDN 资源不是匿名就可以访问并看到的，所以需要订购的客户才能看到并下载。当然，在动态支持中还提供了订阅 MSDN 光盘的方法。

在 Visual Studio 2010 IDE 中，选择“工具”|“选项”命令以后，查看有无“帮助”条目就可以知道你的环境下是否安装了 MSDN。

 **技巧：**作为初学者，需要尽量运用 MSDN 的帮助功能。

1.7 用 Visual Studio 2010 编写第一个程序

在本节中，将通过一个简单的例子，来讲解如何使用 Visual Studio 2010 进行程序开发，

给予读者以直观的认识。

打开 Visual Studio 2010 IDE 之后，新建一个名为 Test 的项目。在 IDE 打开以后，选择“文件”|“新建项目”命令，选择 Windows 窗体应用程序。该文件的默认保存路径是 C:\Documents\Visual Studio 2010\Projects 目录下，如图 1.26 所示，用户可以根据自己需要将项目建立在个人 PC 或者服务器端的指定位置。然后在项目 Test 中，就自动伴随项目生成了一个 Windows 窗体文件 Form1。



图 1.26 新建项目 Test

接下来，固定界面右侧隐藏的“解决方案资源管理器”工具栏，然后在其中单击“查看类图”按钮，视图中出现了如图 1.27 所示的类图，可以帮助用户分析面向对象编程当中各个类之间的关系。

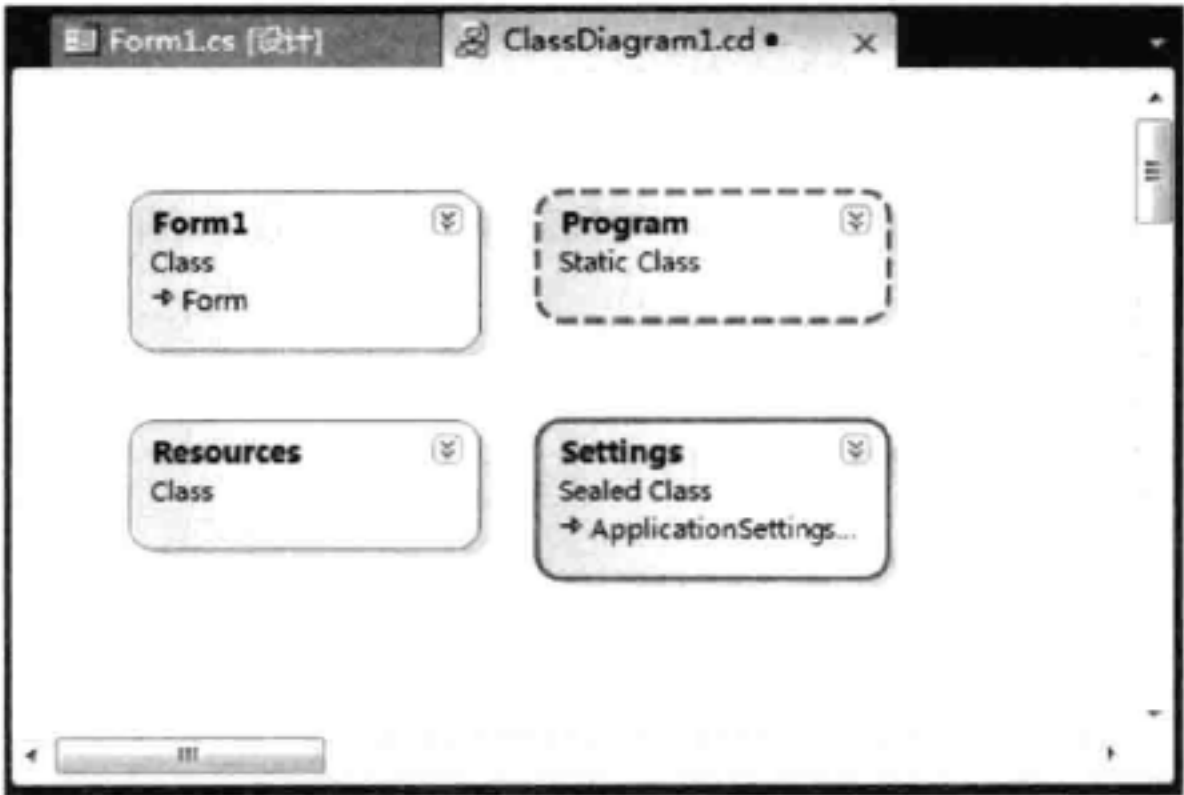




图 1.27 类关系图

然后回到 Form1 的属性页，改变该窗体的部分属性值，具体如表 1.7 所示。

表 1.7 更改属性列表

属 性	值	说 明
Text	练习 1	将 Form1 的显示名称更改为“练习 1”
MaximizeBox	False	取消最大化框
MinimizeBox	False	取消最小化框
BackColor	“系统” “Desktop”	更改窗体背景颜色

更改完属性值以后，选择“生成”|“生成解决方案”命令，一个被修改了属性的简单的由 C#代码编写的 Windows 窗体程序诞生了。按 F5 键，或者通过选择“调试”|“启动调试”命令，就可以生成这个被去掉了最大化与最小化窗口的 Windows 窗口。如图 1.28 所示为调试后的窗口，用户也可以直接单击工具栏区域的  Debug 按钮来执行调试任务。调试通过以后，用户需要单击  按钮，或者单击“调试”|“停止调试”来结束调试状态。

调试通过之后，说明正在编写的程序是正确而健康的，这时候需要进入窗口“练习 1”的代码界面，添加注释。将鼠标停放在窗口练习 1 的任意区域上，然后右击鼠标，选择“查看代码”，在代码的某个部分，分 3 行添加入关键字 TODO、HACK 和 UNDONE。这时，用户可以再次执行“生成解决方案命令”，会发现系统的“错误列表”中出现报错信息。然后用鼠标将这 3 行无效代码全部选中，单击工具栏中的“注释选中行”按钮，则可以一次性地将整段的代码全部注释。这时，选择“视图”|“任务列表”命令，如图 1.29 中任务列表所示，可以看到 3 处用关键字标识注释的相关信息已经全部显示出来。

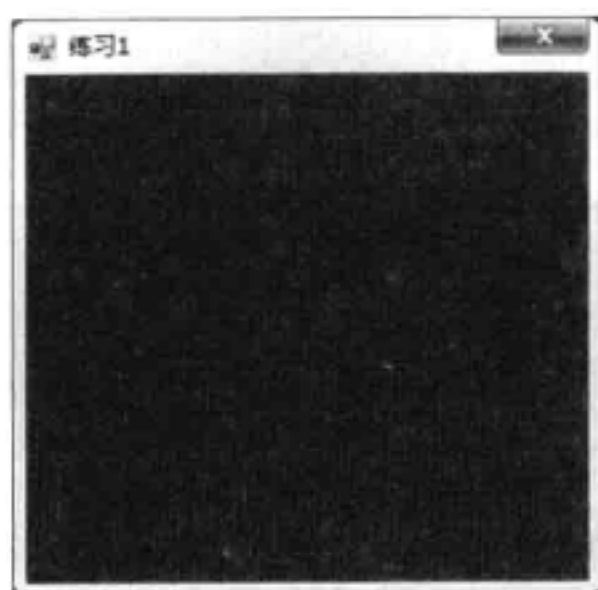


图 1.28 生成“练习 1”窗口

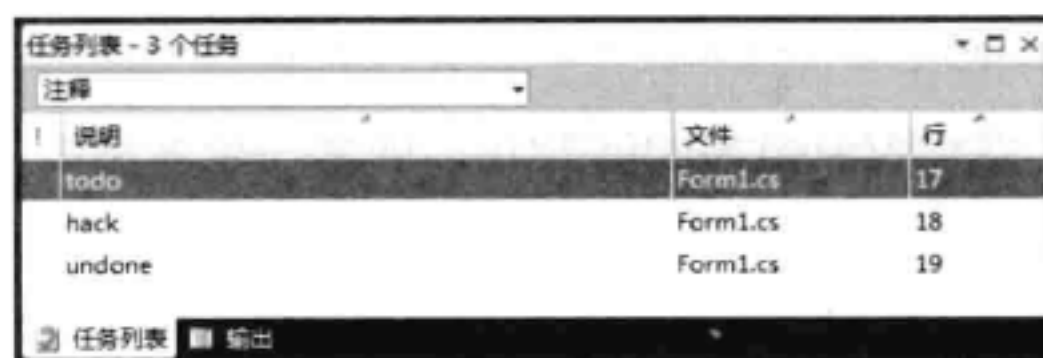



图 1.29 任务列表

 **技巧：**也可以通过单击“取消对选中行的注释”按钮来执行取消大段注释的操作。

读者可以根据本章中介绍的方法自己配置 Visual Studio 2010 IDE 的界面环境，这样能够加深对 Visual Studio 2010 的认识。另外，读者也可以自行对 Test 项目中的这个 Windows 窗体文件 Form1 的各个属性值进行变更或者调解，然后执行编译，看看可以得到什么结果。关于工具箱中控件的使用以及代码部分的知识，将在以后的相关章节中为读者逐个介绍。

1.8 本章总结

本章主要介绍了 .NET Framework、公共语言运行时、中间语言，以及垃圾收集器等基

础概念。在了解了 Visual Studio 家族软件的发展历史以后，带领读者初步认识了 Visual Studio 2010 IDE 的基本环境。掌握了 Visual Studio 2010 IDE 的定制环境的组成与属性的配置方法，同时还进行了一些基本的操作。此外，还帮助读者了解了 MSDN 的概念、作用，以及使用方法。

.NET 平台为广大程序员开发 Application 应用程序提供了一种崭新的途径。许多能够提高开发效力的优势都是显而易见的。举例而言，当开发一个 C++ 桌面应用程序时需要编写许多行的代码。而在 .NET 环境中，实现的代码量可能是相同的，不同之处在于 .NET 通过 BCL 中大量的已经预先创建的类，封装好了这些操作。而非托管代码 (unmanaged code) 是 C++ 的一个成熟的机制，相对于 .NET 平台下的托管代码而言，孰优孰劣很难分辨。然而，在一个完美的模型中，程序员的选择可能是正确的，但是对于当今大量的应用程序开发需求而言，这个就不能成为一个重要的因素了。不断进步的版本支持以及自动存储管理为托管环境增色不少。

.NET 平台的出现及应用是计算机服务连接与开发领域的一次伟大的飞跃。通过 .NET 产品，Microsoft 公司实现了构想中的愿景，这些新的软件产品的开发者如何接受挑战，需要时间的检验和实践的证明。实践证明需要时间的检验。

应该说，Visual Studio 2010 是一个非常强大的平台，在接下来的几章中读者将会逐渐领略其完备的功能，以及卓越的性能。

1.9 实战练习

1. 按本章介绍的方法，将 Visual Studio 2010 安装到计算机中，只选择安装 C# 和 Visual Basic。
2. 启动 Visual Studio 2010，创建一个名称为 First 的项目，选择 Windows 窗体应用程序模板，向项目中添加 3 个窗体。
3. 打开 First 项目，打开 Form1 窗体，向代码中添加 todo 内容。


第2章 C#简介

在 Microsoft 的官方解释中, C# (读做 C sharp) 是从 C 和 C++ 派生来的一种简单、现代、面向对象和类型安全的编程语言。C# 主要是从 C/C++ 编程语言家族移植过来的, 并借鉴了 Java 等面向对象编程语言特点的一种新型的编程语言。使用过 C 和 C++ 的用户会很快地熟悉它。在 .NET Framework 环境下, C# 结合了 Visual Basic (以后简称 VB) 的快速开发能力和 C++ 的强大灵活的能力。本章将为读者具体介绍 C# 语言。

2.1 C#与.NET 的关系

在 Visual Studio 2010 中, C# 是一个专为 .NET 平台设计的新型计算机语言。因此, 在 .NET Framework 中, 对 Visual C# 的支持包括各种文件以及项目模板、设计器、属性页、工具栏、类、对象视图、代码向导, 以及开发环境的其他功能。

C# 是一种简单的、现代的、类型安全的、面向对象的计算机语言, 它使得程序开发者能够快速而容易地在 Microsoft.NET 平台上开发解决方案。Microsoft 公司设计 C# 是为了帮助用户建立运行于 .NET 平台上的、范围广泛的企业级应用程序。Visual C# 代码被编译为托管代码受益于公共语言运行库的服务。

 **说明:** 这些服务主要包括语言互操作性、垃圾回收、增强的安全性, 以及改进的版本支持等。

在 .NET Framework 构架下, 类和数据类型对所有 .NET 语言是通用的。也就是说, 不仅是 C# 语言, 对于 Visual Studio 2010 而言, 包括 VB、Visual C++、F# 在内的语言都可以使用基类库中所封装的类与数据类型。不同之处在于, C# 语言中 Microsoft 公司处理了 C++ 的内存管理以及指针等问题, 这就使得开发人员能够用 C# 语言高效准确地开发控制台应用程序、Windows 应用程序、Web 应用程序等, 而不再会受到兼容性等问题的困扰。

在接下来的几个小节中, 将会为读者具体介绍 .NET Framework 构架下的 C# 语言的特点, 以及与 NET 中其他几种语言的异同。

2.2 C#有哪些特点

作为一种被广泛应用于计算机程序开发的语言, C# 具备很多优越的性能特点。在本节中, 将具体讨论一下这些特点的本质以及实际意义。

2.2.1 简单性

C#的发展与成型都借鉴了很多 C/C++ 的特点,然而与 C 或者 C++ 相比, C# 最突出的特点就在于其语法结构更简单。由于大部分的重复性代码都被封装在了 .NET Framework 的基类当中,所以用 C# 开发应用程序的操作要简单很多。

相信大多数有过程序开发经验的人都知道, C++ 是一种强大的面向对象的计算机语言。在使用 C++ 时,程序员最害怕的是什么呢?大多数人的答案会是:指针!没错,正是由于有了从 C 语言时代继承而来的功能强大的指针,才使得 C++ 脱颖而出。可是也正是由于指针的存在,令 C++ 变得复杂而危险,危机四伏。

在 C# 中,没有了指针。默认地,用户对代码的操作被限定在了一种受控的环境中,语言本身不允许进行直接存取内存等不安全的操作。

除了取消指针以外, .NET 平台下的 C# 语言无时无刻不体现着简约之美。

1. C# 只支持一个“.”

C# 取消了 C++ 中的 “::”、“->” 和 “.” 操作符,对于程序员来说,这就大大简化了对于类以及方法函数操作的复杂性。

2. C# 是基于 .NET 平台的

因此它继承了自动内存管理和垃圾回收的特点,这使得程序本身的构造与析构都更加的方便、容易。

以下这段代码是第 1 章中所建立的一个项目文件中的 Windows 应用程序的代码文件。读者可以很清晰地发现,该应用程序的构造函数非常简单,而且没有已经写好的析构函数。对于一个比较简单的程序而言,析构函数是由 .NET 编译器自动在后台运行的。程序员不需要为释放空间而费心,这样就可以把大量的精力放在代码的其他部分了。

```
//引用的命名空间
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
//本程序所在的命名空间
namespace _2._1
{
    //class Test 继承自 System.Form 类
    public partial class Form1 : Form
    {
        //构造函数
        public Form1()
        {
            //初始化组件
            InitializeComponent();
        }
    }
}
```

```
}
}
```

3. 丰富而简练的关键字

在 C# 中，替换了如 OLE_COLOR、BOOL、VARIANT_BOOL、DISPID_XXXXX 等伪关键字。每种 C# 操作符在 .NET 类库中都有了新名字，这使得 C# 关键字变得简洁易懂。

如表 2.1 所示给出了 C# 中所有的关键字。希望读者能够熟悉它们，因为在以后的学习中将会经常与这些可爱的 keyword 打交道。

表 2.1 关键字

abstract	base	bool
break	byte	case
catch	char	checked
class	const	continue
decimal	default	delegate
do	double	else
enum	event	explicit
extern	false	finally
fixed	float	for
foreach	goto	if
implicit	in	int
interface	internal	is
lock	long	namespace
new	null	object
operator	out	override
params	private	protected
public	readonly	ref
return	sbyte	sealed
short	sizeof	static
string	struct	switch
this	throw	true
try	typeof	uint
ulong	unchecked	unsafe
ushort	using	virtual
void	while	

4. 布尔值仅能为true或者false，不能为0或1

在 C# 中，布尔值只能是纯粹的 true 和 false 值，且不允许更多地使用 “=” 操作符和 “==” 操作符。“==” 操作符被用于进行比较操作，而 “=” 被用于赋值操作。

示例程序如下：

```
//引用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
```



```

using System.Text;
//本程序所在的命名空间
namespace _2._2
{
    class Program
    {
        //主程序入口
        static void Main(string[] args)
        {
            //初始化变量
            char ch1 = 'A';
            string str1 = "A";
            //获得 ch1 的类型
            object c = ch1.GetType();
            //获得 str1 的类型
            object d = str1.GetType();
            //打印 ch1 的类型
            Console.WriteLine("char c=" + c);
            //打印 str1 的类型
            Console.WriteLine("string d=" + d);
            bool equal = (c == d);
            //打印 ch1
            Console.WriteLine("char ch1=" + ch1);
            //打印 str1
            Console.WriteLine("string str1=" + str1);
            Console.WriteLine("两个变量是否相等?" + equal);
            Console.ReadLine();
        }
    }
}

```

运行结果为:

```

char c=System.Char
string d=System.String
char ch1=A
string str2=B
两个变量是否相等? False


```

所有的这些简化都使得 C# 的简单性得以凸现。而简单的操作与语法恰恰是开发型用户的最爱。

2.2.2 类型统一性

除了简单性以外, C# 还具备了类型的统一性。C# 的类型系统是统一的, 所有的类型变量以及值都可以作为对象 (object) 处理。实际上, 这与 C# 本身的类型设计原理有很直接的关系。C# 中每一个类型都可以理解为最终继承自 object 类, 这就意味着 object 是所有类型最终的基类。正是有了这一特性, 使得 C# 中无论值类型, 还是引用类型, 都可以被当作

统一系统类型来对待。


说明：C#提供了装箱（boxing）与拆箱（unboxing）的机制来完成统一类型的属性和方法的操作，而不会给使用者带来任何麻烦。

下面代码为读者展示了类型统一的方法：

```
using System;
class Test
{
    //主程序入口
    static void Main()
    {
        int i = 123;
        //装箱
        object obj = i;
        //拆箱
        int j = (int)obj;
    }
}
```

关于封箱以及拆箱的机制，这里就不详加介绍了。读者只需要理解这种机制对统一类型的帮助即可。有关具体操作的细节，将在后续章节详细讨论。

在后面的学习中读者们会发现，C#中取消了全局函数、全局变量和全局常量这一类的概念。所有的一切，都必须封装在一个类之中。正是由于C#的每种类型都可以看做一个对象，才使得其代码将具有更好的可读性，并且减少了发生命名冲突的可能。

说明：由于在C#中只允许单继承存在，即一个类不会有多个基类，这避免了类型定义的混乱，也确保了类型统一的合法性。

2.2.3 面向对象性

对于程序员而言，从Smalltalk开始，面向对象的话题就始终围绕在所有现代程序设计语言周围。此后，一种新语言不支持面向对象的功能是不可想象的。C#支持所有关键的面向对象的理论与概念，并且C#具有面向对象语言所应具备的一切特性，能实现封装、继承与多态性。然而，作为一门彻底的面向对象的语言，C#比之前的C++或者Java具备更加完备、合理的面向对象结构，可以满足各种层次规模的开发需求。

C#的面向对象性拥有其自身的一些特点。作为一门独创的新型语言，C#的类模型是建立在.NET虚拟对象系统（Visual Object System, VOS）的基础之上，这就使得它的对象性得到了升华。正是由于C#的对象模型仅是.NET Framework基础架构的一部分，而不再是其本身的组成部分，所以降低了以往面向对象语言中所存在的面向过程性。这些令它的面向对象性在以下几方面得到了很好的体现。

1. C#支持数据封装、继承、多态和接口

作为一门面向对象的语言，以上这些特性都是C#所必备的。关于这些特性的具体体现，将会在以后的相关章节中慢慢讨论。

2. C#的最底层数据类型为对象（object）

C#不仅通过引入结构体(struct)而使得函数段落得以被封装,甚至连 int、float 和 double 这些在 Java 中都不是对象的常量在 C# 中也成为对象。

最为经典的例子如下:

```
using System;
class Test
{
    //主程序入口
    static void Main()
    {
        int i = 1;
        //转换(或者)Boxing, 可以得到一个字符串“”
        String a = i.ToString();
    }
}
```

3. C#中没有真正意义上的全局函数、变量或者是常量

以往, C++的处境就显得非常尴尬。熟悉的读者可能知道, 在 C++程序中, 人们无法解释 main 函数属于哪个类, 但 C#却实现了真正纯粹的面向对象。

由于 C#具有一流的面向对象的设计, 使得用户从构建组件形式的高层商业对象到构造系统级应用程序都可以选择它来完成开发。结合自身强大的面向对象功能, C#具备了良好的开发环境。对于程序员而言, C#能够极大地提高开发人员的生产效率。而对于企业而言, C#面向对象的彻底性可以缩短软件开发周期。这使得一切都完美地结合了起来。

2.2.4 类型安全性

语言的安全性是衡量一种语言是否优秀的重要标准。即使是最熟练的程序员也会犯错误, 谁都不能避免可能会忘记变量的初始化, 也许会不小心对不属于自己管理范围的内存空间进行修改等。这些错误常常会产生难以预见的后果。一旦这样的软件被投入使用, 人们将会投入巨大的代价来寻找与改正这些简单错误, 这将会让用户损失巨大。


C#的先进设计思想可以消除软件开发过程中的许多常见错误, 并提供了包括类型安全在内的完整的安全性能。为了减少开发中的错误, C#在.NET Framework 的帮助下可以令开发者通过更少的代码完成相同的功能。

C#对于安全性的控制主要体现在以下几方面:

- ☐ 利用.NET 运行时可以控制代码访问的安全特性;
- ☐ 在 C#中变量类型是安全的;
- ☐ 在 C#中类型转换是安全的;
- ☐ 在 C#中数组类型进行越界检查;
- ☐ 编译器会提醒未经初始化而被使用的局部变量;
- ☐ C#语言检查类型溢出。

所有的这些都确保了 C#的类型安全性得到很好的控制。.NET 平台提供的垃圾收集器 (Garbage Collection, GC) 将负责资源的释放与对象撤销时的内存清理工作, 这些工作都

是自动执行的，不再需要程序员亲自指定或者排除。所有这些都大大地提高了 C#语言的安全性。

 **注意：**利用垃圾收集机制管理内存减轻了开发人员对内存管理的负担。

2.2.5 兼容性


一种新语言的出现，总是要考虑对之前的技术体系是否兼容的问题。这里必须提到风靡全球的另一种语言——Java。它在出现伊始就彻底背弃了之前处于统治地位的 COM 技术体系，尽管 Java 语言得到了用户的认可，然而时至今日它仍很难去瓦解 Visual C++（以后简称 VC）、VB 和 Delphi 等阵营。或许正因为如此，促使它只能在此之外的其他技术和市场领域扎根。

.NET Framework 实现了对以往几乎所有主流技术的兼容。可以看到，越来越多的 VC、VB、Delphi 的程序员们转到了 VC.NET、VB.NET 和 Delphi.NET。作为 .NET Framework 的首推语言和最具特色的独创语言，C#在简化语法的同时，不仅并没有失去灵活性，并且很大程度上保持了对外界技术的兼容。

C#的兼容性主要体现在以下几个方面：

- ☐ C#提供对 COM 和基于 Windows 应用程序的原始的支持；
- ☐ C#在特定环境下允许有限制地使用原始指针；
- ☐ C#允许用户使用指针来操作 C++的代码，但前提是必须声明是不安全的；
- ☐ C#类库中已经内建 `unkown` 和其他 COM 界面，用户不再需要显式地实现这些功能；
- ☐ .NET Framework 平台下的（如 VB.NET 等）所有中间代码语言中的组件可以在 C#中直接使用。

在用户需要的某些状况下，C#允许将某些类或者类的某些方法声明为非安全的。因此，用户将能够使用指针、结构和静态数组，并且在调用这些非安全代码的过程中不会带来任何其他的问题。

 **注意：**C#还提供了委托（delegate）来模拟指针的功能。C#不支持类的多继承，但是用户可以通过对接口的继承来实现类似的功能。这些都很好地体现了 C#与 C++，以及 Java 语言的兼容性。

2.3 C#与其他语言对比

通过第 1 章的学习读者已经知道，在 Visual Studio .NET 里，.NET Framework 和 CLR 是所有应用程序运行的基础。无论使用的是哪一种语言，在 .NET 这个平台上都将编译成 Microsoft 中间语言（即 MSIL），以达到无缝集成的目的。

C#、C++，以及 Java 都是享用共同的基础，也就是所谓的“根结构”。因此与其他计算机语言相比他们之间有很多的共同点。然而，VB 与这三种语言是有一定区别的，但是 VB 依然分享了许多相同的语法要素。

本书将通过代码实例,进一步地对.NET Framework 平台下的各种语言展开比较,以使读者能更加清楚地了解C#的特点与优势所在。

2.4 C#与VB.NET的异同

C#与VB是两种完全不同的语言。C#是一种面向对象的计算机语言,而VB仅仅具有非常有限的一些面向对象的特点。从VB 6到VB 7,尽管期间VB增加了一些面向对象的因素,但那只是对于学习使用文档有所帮助。

VB.NET与传统Basic语言相比,具有许多新功能和改进功能,如继承、接口和重载,自由线程处理和结构化异常处理等。这些让VB.NET成为功能强大的面向对象的编程语言。

C#与VB.NET有很多的相同或者相似之处,但是它们最一致的特点就是:

- VB.NET和Visual C#都具有RAD(快速应用程序开发)支持,以及项目模板、设计器和其他的开发环境功能。
- VB.NET和Visual C#这两种语言都使用.NET Framework基类。

当然,由于这两种语言之间的联系与区别有着千丝万缕的关系,不是一言可蔽之的。所以从功能上看,C#的确有VB.NET无法完成的功能,但VB.NET也有C#无法完成的功能。总地来说,对于.NET开发,使用VB.NET或者Visual C#没有任何的差别,它们都能够完全地与CLS兼容。

在下面的各节中,将归纳总结C#与VB.NET的主要异同之处。

2.4.1 代码表现形式的差异


在VB.NET中,语句块是以END语句来结束,而且语句的书写格式有非常严格的限制,不允许在一行中书写超过一条的复杂语句。在C#中,语句块使用大括号{}表示,而且对于结束行的位置没有非常具体的限制,只要使用分号标识就可以了。尽管相对于VB.NET而言,C#的格式可能会显得有些糟糕,并且容易产生不利于阅读的状况,如下所示。

```
for (int j = 0; j < 100; j++)  
{  
    if (j == 10)  
        Func(j);  
    Else  
        return;  
}
```

很明显,上面的代码看起来不是很舒服,而且逻辑上也显得杂乱无章。但是请不要怀疑,在语法上它是完全正确的。实际上,这段代码如果按照逐行的书写模式就变成以下的状态了:

```
for (int j = 0; j < 100; j++)  
{  
    if (j == 10)
```

```
    Func(j);  
    else  
        return;  
}
```

 **技巧：**尽管代码格式并不是一门语言中最重要的特性，但是有时候对于规范开发，以及代码的可读性都有很大的帮助。

2.4.2 数据类型和变量使用的差异

尽管在数据类型方面，VB.NET 与 C#之间有相当多的重叠，然而还是有一些重要的差异。在这两种语言间，一个相同的类型名称可能意味着完全不同的数据类型。C#当中最重要的不同就是在变量的声明以及使用方面非常严谨。所有的变量都必须在使用前被预先声明，而且他们必须被声明为一种特有的类型。这里不存在能够支持任何类型的 **Variant** 类型。

利用在使用变量前确定数据类型的方法，变量的声明变得非常简单，但是 C#取消了 **dim** 语句。

1. 类型转换

在 C#中，类型之间的转换也要比 VB.NET 严格得多。C#有两种转换类型，即显式转换和隐式转换。隐式转换不会丧失数据，这种转换方法是建立在源类型适用于目标变量的基础上的。示例如下：

```
int x = 15;  
long y = x;
```


将 **x** 转化为 **y** 是允许的，因为 **int** 型变量能够适应 **long** 型变量的域值设置。从另一个方面来看，显式转换是一种会造成数据丢失或者转换失败的转换方式。也正是由于显式转换的这个特性，所以必须用一定的方法来明确转换双方的身份。示例如下：

```
long y = 15;  
int x = (int) y;
```

在这个示例中，转换是安全的，尽管 **long** 型的变量比 **int** 型变量存储数据的位数要大的多，但是代码在两种类型转换之前已经明确声明了转换的类型，也就完成了转换。

2. 数据类型的区别

在 VB.NET 中，完备数据类型包括 **Integer** 和 **Long**。在 C#中，这些已经被 **short** 型与 **int** 型所替代。在 C#中也有 **long** 型数据，但是它是 64bit 的数据类型。VB.NET 中的字节型 **Byte** 型被重新命名为 **byte**。

 **技巧：**当 C#中的 **long** 型与 VB.NET 中的 **Long** 型替换时，程序将会被编写得更为庞大，运行速度将会降低。

C#中也有无符号数据类型，诸如 **ushort**、**uint** 和 **ulong** 等。也包括有符号的字节类型 **sbyte**。这些数据类型在某些特定环境中都是非常有用的。但是，它们不能被 .NET 的其他语言所使用。因此，它们仅在必要的场合下才能使用。

VB.NET 中的 Single 和 Double 类型在 C# 中也都重新命名为 float 和 double 类型，而且 VB.NET 中的 Boolean 型被精简为 C# 中的 bool 型。

3. 字符串

许多在 VB.NET 中已经存在的固有函数对于 C# 中的 string 类型而言都是不存在的。C# 中有专门的函数进行查找字符串，提取字符串中的字节，还可以进行其他的操作。

C# 中的 string 类型可以使用加号 “+” 来连接，而不再使用 VB.NET 中字符串连接号 “&” 操作符。

4. 数组

在 C# 中，数组的第一个元素的索引号是 0，没有任何方法可以扩展或者缩减数组的范围，并且没有 VB.NET 中的 redim 关键字来修饰一个数组。但是，所谓的动态数组 ArrayList 却是在 System.Collection 命名空间中，并且允许自定义数组大小。

2.4.3 类、数据类型、函数以及接口

由于 C# 是一种面向对象的计算机语言，所以类的概念就成了这门语言最为主要的组成单元。甚至，用一些特有的类代替了传统意义上的全局函数和全局变量。这样的结果就使代码的结构和组织性都与 VB.NET 有非常大的不同，但是这两种语言之间还是存在着一些共同的基础。

在 C# 中，属性依然能够被使用，只是它们有了不同的语义，而且不再有默认属性了。

此外，在 C# 中，函数所携带的参数必须有一种声明的类型，而且 ref 的应用取代了原来 VB.NET 中 ByVal 的位置用来指出一个已经通过的变量值是固定的。使用 params 参数就可以实现 VB.NET 中 ParamArray 函数能够实现的一切功能。

2.4.4 操作符与表达式的差异

C# 和 VB.NET 的操作符也有很多不同之处，它们的表达式还是有一些相同的语法。具体的区别见表 2.2。

表 2.2 C# 与 VB.NET 操作符对比表

VB.NET	C#
Mod	%
&	&
=	==
<>	!=
And	&&
Or	
Xor	^
^	没有。用 Math.Pow() 方法取代
Like	没有。System.Text.RegularExpressions.Regex 可以实现类似的操作，但是其本身的功能要更加复杂

续表

VB.NET	C#
Is	没有。实际上，C#中的 is 操作符代表的意义完全不同
Eqv	没有
Imp	没有

2.4.5 控制流程语句的差异

C#与 VB.NET 有着十分相似的控制语句，但是在语法上有些不同。

1. If Then

在 C#中，没有 VB.NET 的 Then 语句，当条件判断结果为真时，程序直接执行之后的语句或者语句块，紧接其后的是另一个可选分支 else 语句。

VB.NET 的代码如下：

```
If size < 100 Then
    value = 10
Else
    value = 15
    order = 20
End If
```

在 C#中，代码如下：

```
if (size < 100)
    value = 10;
else
{
    value = 15;
    order = 20;
}
```

另外，在 C#中没有 VB.NET 中的 ElseIf 语句。

2. For

与 VB.NET 相比，C#中的 for 循环在语法上是不同的。但是，除了在 C#中必须明确地指定每一个循环的结尾之外，两种语言的 For 语句的概念是一致的。

VB.NET 的示例代码如下：

```
For i = 1 To 100{
    ' 函数体
}
```

C#中代码如下：

```
for (int i = 0; i < 100; i++)
{
    // 添加函数体
}
```

3. For Each

C#支持 foreach 语法，相当于 VB.NET 中的 For Each 语法。foreach 可以应用于数组、收集器类和其他有适当接口的各种类。

4. Do Loop

C#有两种循环结构来替代 VB.NET 中的 Do Loop 结构。while 语句用于循环条件判断为真（true）的条件。而 do while 语句的循环过程也是一样的，不同之处是当条件判断为假（false）时，仍然可以运行一次循环。

VB.NET 代码如下：

```
i = 1
count = 1
Do While i <= 199
    count = count * i
    i = i + 1
Loop
```

C#代码如下：

```
int i = 1;
int count = 1;
while (i <= 199)
{
    count = count * i;
    i++;
}
```

⚠注意：C#中的循环需要利用 break 语句才能退出，或者利用 continue 语句继续下一个重复的过程。

5. Select Case结构

C#中用 switch 语句替代了 VB.NET 中的 Select Case 结构。

VB.NET 中代码如下：

```
Select Case x
Case 1
    Func1
Case 2
    Func2
Case 3
    Func2
Case Else
    Func3
End Select
```

C#中代码如下：

```
switch (x)
{
    case 1:
        Func1();
        break;
```

```

case 2:
case 3:
    Func2();
    break;
default:
    Func3();
    break;
}

```

2.4.6 错误处理的差异

C#中没有 VB.NET 中的 On Error 错误处理语句。.NET 中的错误条件与异常处理机制是紧密相连的。有关细节将会在以后的章节中逐步详细讲解。

2.4.7 关键字的差异

VB.NET 的关键字一般比 C#的要长一些，而且使用限制也更加严格。表 2.3 中列出了它们的关键字的对应关系。

表 2.3 C#与VB.NET关键字的对应关系


VB.NET	C#	VB.NET	C#
Imports	using	Friend	internal
Me	this	Shared	static
MyBase	base	Sub	void
MustInherit	abstract	Select	switch
MustOverride	virtual	(Of T)	<T>
NotOverridable	sealed		

2.4.8 访问修饰符的差异

VB.NET 与 C#的访问修饰符也存在一些异同。具体列表如表 2.4 所示。

表 2.4 访问修饰符

	VB.NET	C#	意 义
类	Public	public	访问不受限制
	Friend	internal	只在程序集内部可访问
方法函数	Public*	public	不受限制
	Friend	internal	只在程序集内部可访问
	Protected	protected	访问仅限于包含类或者从包含类派生的子类
	Protected Friend	protected internal	访问仅限于类以及从类派生的子类中，或者当前程序集的其他类
	Private	Private	访问仅限于包含类

 注意：在 VB.NET 中，通过关键字 Dim 声明的方法默认为 Public，而字段默认为 Private。

2.4.9 语法的差异

实际上，VB.NET 的语法要比 C# 丰富的多。在 VB.NET 中，通过 Events、Try...Catch 和 Select...Case 等语句可以直观地实现相应的功能，但是面向对象性会显得薄弱一些。C# 取消了很多面向过程性语法，取而代之的是全部用类的封装等技术来实现。具体对比如表 2.5 所示。

表 2.5 VB.NET与C#语法异同

	VB.NET	C#
动态数组	有	没有
模 (modules)	有	没有
可选参数	有	没有
内置的插入点支持	有	没有
移位操作符	没有	有
重载操作符	没有	有
内嵌的文档 (XML)	没有	有
Interface 后期绑定 (Late binding)	能实现	不能
参数属性 (parameterized properties) 后台编译	执行	不能

2.4.10 C#与 VB.NET 实例对比

通过以上的学习，相信读者已经能够掌握 VB.NET 与 C# 两种语言的异同之处了。在 .NET 平台下，两种语言的可移植性都得到了提高。在熟悉了 C# 以及 VB.NET 的简单语法以后，就可以在 .NET 平台下实现两种语言的有效移植。此外，这两种语言还可以通过特定的代码转换工具进行移植转化。下面，以两种语言的主入口函数的对比为例，来加深读者对它们的认识。

在实现 VB.NET 的 Main 函数时有以下几种方法：

```
Shared sub Main() : end sub
Shared sub Main(argv as string()) : end sub
Shared function Main() : return 0 : end function
Shared Function Main(ByVal argv As String())
    Return 0
End Function
```

在实现 C# 的 Main 方法时有以下 4 种方法。

```
static void Main() { }
static void Main(string[] argv) { }
static void Main() { return 0; }
static void Main(string[] argv) { return 0; }
```

这里必须指出，在 MSDN 文档中，C# 和 VB.NET 的地位是旗鼓相当的。也就是说，它们的地位与受重视程度是一样的。所有文档和示例都是分别由 VB.NET 和 C# 两种语言组成的。因此读者在学习过程中不用担心资源的问题。

下面以最为典型的事件机制为例，为读者分析一下 VB.NET 和 C# 的不同。
在 VB.NET 语言中，事件机制的声明与调用过程如下：

```
''' <summary>
''' 声明代理
''' </summary>
''' <param name="sender"></param>
''' <param name="e"></param>
''' <remarks></remarks>
Delegate Sub myEvnetHandler(ByVal sender As Object, ByVal e As EventArgs)
''' <summary>
''' 创建事件发布者类
''' </summary>
''' <remarks></remarks>
Class Release
    Public Event Event1 As myEvnetHandler
    Public Sub DoEvent1()
        RaiseEvent Event1(Nothing, Nothing)
    End Sub
End Class
''' <summary>
''' 创建事件接收者类
''' </summary>
''' <remarks></remarks>
Class Receive
    Public Sub New(ByVal rls As Release)
        AddHandler rls.Event1, AddressOf OnEvent1
    End Sub
    Sub OnEvent1(ByVal sender As Object, ByVal e As EventArgs)
        Console.WriteLine("启动 VB 事件")
        Console.ReadLine()
    End Sub
End Class
''' <summary>
''' 实例化事件发布者类、接收者类，并引发事件
''' 事件只能由发布者调用，接收者注册
''' </summary>
''' <remarks></remarks>
Module Module1
    Sub Main()
        Dim R As Release = New Release()
        Dim C As Receive = New Receive(R)
        R.DoEvent1()
    End Sub
End Module
```

C#代码如下：

```
using System;
```



```

using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _2._4
{
    /// <summary>
    /// 声明代理
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    delegate void Event1Handler(object sender, EventArgs e);
    /// <summary>
    /// 创建事件发布者类
    /// </summary>
    class Release
    {
        public event Event1Handler Event1;
        public void DoEvent1()
        {
            if (Event1 != null)
            {
                Event1(null, null);
            }
        }
    }
    /**/
    /// <summary>
    /// 创建事件接收者类
    /// </summary>
    class Receive
    {
        public Receive(Release rls)
        {
            rls.Event1 += new Event1Handler(rls_Event1);
        }
        void rls_Event1(object sender, EventArgs e)
        {
            Console.WriteLine("启动 C# 事件");
            Console.ReadLine();
        }
    }
    /**/
    /// <summary>
    /// 实例化事件发布者类、接收者类，并引发事件
    /// 事件只能由发布者调用，接收者注册
    /// </summary>

```





```

class Program
{
    //定义入口点 Main() 函数
    static void Main(string[] args)
    {
        Release R = new Release();
        Receive C = new Receive(R);
        //调用 R 的 DoEvent1() 方法
        R.DoEvent1();
    }
}

```

这两个程序分别实现了事件定义及触发，其中涉及了比较复杂的背景知识，读者如果没有编程经验，也不必一定要看懂代码各段的意义与功能。这里展示这两段代码最主要的目的是让读者能通过以上代码的比较，把本小节中所列出的两种语言的众多不同之处做一番具体的了解。

 **注意：**由于事件触发机制不同，所以两种语言中的代码有一定的差别。

 **说明：**读者不用了解这两种语言代码不同之处的具体缘由，仅需要在了解 VB.NET 的前提下掌握一些它们的差异所在就可以了。

2.5 C#与 C++的异同

C#代码与 C 和 C++程序非常相似，但是它们之间也存在着极少部分有巨大差异的地方，以及许多细微的不同之处。接下来，本节将这些不同之处进行对比总结。

2.5.1 托管环境的差异

通过第1章的讲解，相信读者已经清楚地了解到 C#在.NET 运行时 (.NET Runtime) 环境中运行。这不仅意味着有许多部分不在程序员的控制之下，同时也意味着系统提供了一系列的新标志性构架。两者结合的结果就是改变了一些要素。

- ☐ 当对象不再被使用时，清除对象实例的工作由垃圾收集器自动执行。
- ☐ C#语言中不再有指针。但是在某些特定的环境下，还允许有限地使用它。取而代之的是引用参考，这与 C++中语法定义很相像，只是取消了很多 C++中的限制。
- ☐ 用控制异常 (Exception) 取代了错误 (Error) 返回机制。
- ☐ 不再有类似 C/C++中的运行时库。

在 C#中，用.NET 运行时，以及命名空间 (namespaces) 取代了原来的 I/O 文件夹和字符串处理等。

2.5.2 C#使用.NET 的对象

C#的对象全都是继承自一个共同的 `object` 类，而且所有的类只允许单继承。当然，可以通过接口来实现实际意义上的多继承。当对象需要封箱操作时，C#中的结构和其他的值类型能够在一些条件下被使用。

2.5.3 C#的语句

C#的语句高度地忠实于 C++ 的语句构成及语法。仅有一些不同的地方，列举如下：

❑ `new` 关键字意味着“获得一个新的复制值”。

当作为引用类型时对象是一种被分配在堆（`heap`）上的类型，而值类型则被分配在栈（`stack`）上。

❑ 所有的语句用一个新的 `bool` 类型变量来进行布尔值的判断。

❑ 为了减少错误，`switch` 语句不允许无判断遍历，但是 `switch` 语句也可以使用 `string` 类型的值。

❑ 新引入的 `foreach` 语句可以用来重复遍历对象集。

❑ `checked` 以及 `unchecked` 语句被用来控制数学运算的溢出检查。

2.5.4 C#中取消的要素

❑ 多重继承；

❑ `const` 成员函数或者参数，支持 `const` 局部变量；

❑ 全局变量；

❑ `Typedef`；

❑ 函数参数中的 `default arguments`。

2.5.5 操作符重载的差异

C++与C#都可以执行操作符重载。C#中能够被重载的操作符只是部分，而且所有的赋值运算符都不能被重载。具体如表 2.6 所示。

表 2.6 C#重载运算符


单目操作符	二元运算符	单目操作符	二元运算符
+	+		<<
-	-		>>
!	*		==
~	/		!=
++	%		>
--	&		<
True			>=
False	^		<=

2.5.6 头文件的差异

在 C++ 中, #include 包含语句可以指明所有被调用的头文件物理路径, 完成引用。而 C# 则利用 using 指令编译器程序将在 System 命名空间下操作, 所有的命名空间和类都属于 System 命名空间。

2.5.7 程序书写的差异


在 C++ 中, 无论程序书写还是变量的定义都是不区分大小写的。而 C# 中大小写的区分十分重要。C# 的主程序 Main 必须严格区分大小写。

 **技巧:** 在 C++ 中, 类声明结束后必须要在最后一层大括号后再添加一个分号以结尾, 否则将会编译报错; 而 C# 则没有这个规定, 实际上任何大括号的后面都是不允许紧跟着添加符号的。

2.5.8 被取消的指针

C# 对于 C++ 而言有一定的继承性, 因此它们在很多语法结构上非常相似。二者最显著的区别就在于 C# 不再允许指针存在。正是由于这点, 使得 C# 的语言安全性被大大提高。

在 C# 中, 取代指针的是一种全新的类型——委托 (delegate)。相对于指针而言, 委托的功能更为强大。

 **说明:** 委托可以在方法之间传递, 并且可以包含所调用类中方法实例的信息。

2.5.9 虚函数的差异

C# 与 C++ 都支持虚函数的概念, 但是 C# 的虚函数则更加安全。

尽管 C++ 与 C# 对于虚函数的实现方式相同, 但是在 C# 里实现同样的功能要更容易。下面就通过具体的代码来了解这两种语言所支持的虚函数的不同之处。

C++ 代码如下:

```
//头文件
#include <iostream>
class Base
{
    public:
    void aFunction()
    {
        cout << "正在运行基类";
    }
    protected:
    //声明纯虚函数
    virtual void aFunction1() = 0;
```



```
};
//继承基类
class Derived : public Base
{
    public:
    void aFunction2()
    {
        //输出文本
        cout << "正在运行自定义的类";
    }
    void aFunction1()
    {
        cout << "正在运行虚函数";
    }
};
//主程序入口
void main()
{
    Derived d;
    d.aFunction1();
}
```

在上面的代码中共有两个类，分别称为 Base（基类）和 Derived（继承类），基类里有两个 void 函数，一个是 aFunction()，另一个是纯虚函数 aFunction1()。aFunction1() 只能被基类的继承类使用。在继承类里有一个新函数 aFunction2() 和继承于基类纯虚函数的超越函数 aFunction1()。

C#代码如下：


```
//引用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//命名空间
namespace _2._5
{
    class Program
    {
        //主程序入口
        static void Main(string[] args)
        {
            Derived d = new Derived();
            //aFunction1() 与纯虚函数同功能
            d.aFunction1();
        }
    }
    //定义基类为抽象类
    abstract class Base
    {
        public void aFunction()
        {
            Console.WriteLine("正在运行基类");
        }
        //定义 aFunction1() 为抽象方法
        public abstract void aFunction1();
    }
    class Derived : Base
```

```

{
    public void aFunction2()
    {
        Console.WriteLine("正在运行自定义的类");
    }
    //对 aFunction1 函数进行重写
    public override void aFunction1()
    {
        Console.WriteLine("正在运行虚函数");
    }
}
}

```

为了让继承类能直接操作基类成员和方法，C#为基类命名了一个别名 `base`。用这个别名，继承类可以直接引用基类成员和方法。C#将基类定义为抽象类，将 `aFunction1()` 定义为抽象方法。这样就可以获得与上述 C++ 纯虚函数同样的功能。Base 类只能作为基类或被含有 `aFunction1()` 超越函数的继承类使用。

 **说明：**继承类在超越抽象函数 `aFunction1` 时，在函数前面加上超越前缀(`override`)。C# 编译器在发现继承类里的 `override` 关键字后，就检查基类的同名函数。只要不是直接显式地调用基类函数，编译器总是使用继承类中的方法。

在 C++ 中的参数引用传递表达为：`void ShowType(Person& p)`。

在 C# 中的参数引用传递表达为：`void ShowType(ref Person p)`。`ref` 关键字告诉 C# 编译器向 `ShowType` 函数传递一个参数的引用 (`reference`)。在 C# 中，如果不用 `ref` 关键字，函数的参数传递默认为值 (`value`) 传递，将复制一个参数值传递到函数中。

2.5.10 C#与 C++实例对比

作为面向对象语言，在这里仅以如何建类并进行实例化来比较一下 C++ 与 C# 的不同。C++ 语言版本如下：

```

#include <iostream>
class MyClass
{
    public: void towork()
    {
        //输出字符串 This is a test text
        std::cout << "This is a test text";
    }
};
//C++中必须具有的主函数 main()
void main()
{
    MyClass mc;
    mc.towork();
}

```

C# 语言代码如下：

```

using System;
using System.Collections.Generic;
using System.Linq;

```



```

using System.Text;
//命名空间
namespace _2._5
{
    class Program
    {
        static void Main(string[] args)
        {
            //利用 new 关键字为 MyClass 类的实例 mc 分配空间
            MyClass mc = new MyClass();
            //调用自定义的 towork() 方法
            mc.towork();
        }
    }
    class MyClass
    {
        public void towork()
        {
            //输出文本
            Console.WriteLine("This is a test text");
        }
    }
}

```

2.6 C#与 Java 的异同

C#与 Java 有着十分相似的本质联系，因此它们之间的相似性非常多也就不足为奇了。实际上，这两种语言之间还是有一些不同之处的。它们最大的不同在于 C#是建立在.NET Framework 和运行时（Runtime）基础之上的，而 Java 是建立在 Java Frameworks 和运行时之上的。


在下面的各节中，将归纳总结 C#与 Java 的主要异同之处。

2.6.1 数据类型的差异

C#拥有比 Java 更多的原始数据类型。如表 2.7 所示，总结了两种语言中数据类型的异同。

表 2.7 C#与Java数据类型对比

Java	C#	Java	C#
byte	sbyte	string	string
short	short	object	object
int	int	无	byte
long	long	无	ushort
Boolean	bool	无	uint
float	float	无	ulong
double	double	无	decimal
char	char		

说明：在 Java 中，原始对象类型和基于对象的类型属于不同的世界。由于原始数据类型可以加入到基于对象类型的世界，所以它们必须被封装入实例当中，而且封装类会被置入收集器。

C#解决这类问题的方法是不同的。在 C#中，原始类型如同 Java 中一样是存储在栈中的，但是它们也同样可以理解为是来自最原始的基类——object。这也就意味着原始类型能够定义成员函数，并且可以调用它们。也就是说，下面的示例程序是合法的。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//命名空间
namespace _2._6
{
    class Program
    {
        //主程序入口
        static void Main(string[] args)
        {
            Console.WriteLine(5.ToString());
        }
    }
}
```

2.6.2 类的差异

C#中的类与 Java 中的类非常相似，接下来就重点讲解它们之间的几点重要的不同之处。

1. 常量

Java 使用 `static final` 语法来声明一个类的常量，而 C#则用 `const` 来执行同样的操作。在 C#中，还增加了 `readonly` 关键字以适应编译时常量值不确定的情况。`readonly` 常量的作用域仅能在类的构造函数中或者特别设定。

2. 基类与构造函数

C#使用 C++的语法来定义基类以及类的接口，以用来调用其他的构造函数。一个 C#的类应该如下列代码所示。


```
public class Object1: Class1, IFormattable
{
    public Class1(int value)
    {
        this.value = value;
    }
    public Class1() : base(value)
    {
    }
    int value;
}
```

3. 静态构造函数

与Java中使用静态初始化程序块不同，C#提供了静态构造函数。利用static关键字就可以实现Java中的相似功能。

4. 虚函数

在C#中，所有的方法都是默认为非虚的，virtual必须要在一个函数被直接定义为虚函数的时候才能使用。

说明：C#中没有所谓的final方法，尽管利用sealed关键字也可以实现final类的所有功效。sealed声明的是一个非抽象的类，但它也不能被用做另一个类的基类。

C#能够提供比Java更为人性化的程序界面，但其实质还是有一些改变的。重载方法实际上是对名字的操作而不是对标识符。这就意味着，对于基类添加的新增类将不会改变编译行为。实例代码如下：

```
public class A
{
}
//B 继承 A
public class B: A
{
    public void Process(object obj) {}
}
class Test
{
    public static void Main()
    {
        A a = new A();
        //执行调用
        a.Process(12);
    }
}
```

2.6.3 属性定义的差异

相信大家对属性的概念应该并不陌生。在面向对象编程的语言里，类成员函数可以自由地访问本类中的任何属性成员。不过，如果要从一个类中去访问另一个类中的属性，那就比较麻烦了，Java代码是这样的：

```
public int getSize()
{
    return size;
}
public void setSize (int value)
{
    size = value;
}
```


但是，在C#中，这样的方法被“属性化”了。同样的代码，在C#就变成了：


```

public int Size
{
    get
    {
        return size;
    }
    //get, set 必须配对出现
    set
    {
        //变量 size 的 value 值
        size = value;
    }
}

```

可以看出，C#显然更容易阅读和理解。

说明：为了区分这种属性化的方法和类的属性成员，在C#中把属性成员称做“域(field)”，而“属性”则成为这种“属性化的方法”专用的名词。

2.6.4 事件、指针与界面的差异

C#对事件是直接支持的，可以直接使用 `delegate` 和 `event` 关键字来解决这个问题。而Java用改编类来实现。C#与Java都消除了指针，不过在注明 `unsafe` 关键字的前提下，C#中还是允许在特定条件下使用指针的。

C#与Java相同都可以利用一个单一的类去实现几个界面，也可以采用一个结构体去实现界面。也许在细节方面C#程序会出现一些微妙的差别，但是这个特点看起来与Java相比没有实质的变化。

2.6.5 C#与Java实例对比

通过以上比较，相信读者已经能够了解这两种语言的主要相同点及不同点了。下面显示Hello World内容的程序为例来说明两种语言的区别。

在Java语言环境中代码如下：

```

import java.io.File;
public class ClassHello
{
    public static void main(String[] args)
    {
        System.out.println("Hello Java World!");
    }
}

```

Java的关键字 `import` 与C#中的 `using` 关键字的效用一致。它们在程序中起到了同样的作用。在C#中类的声明与Java很相似。

C#语言程序代码如下：

```


using System;
class Class1
{
    static void Main()

```



```
{  
    //命名空间 System 中的 Console 类的 WriteLine() 方法  
    System.Console.WriteLine("Hello, World");  
}
```

C#与Java之争一直在进行着。Java程序必须在安装了Java虚拟机或者JVM的任何平台上运行，而.NET程序的平台则要求安装通用语言运行库CLR。对于不同的用户，针对开发项目的不同需求，可以选择自己擅长的语言。

说明：在Visual Studio 2010平台下，C#的强大功能是非常突出的。

2.7 本章总结

本章主要讨论的就是.NET版本下的Visual C#与VB.NET、Visual C++，以及Java语言之间的异同。尽管在.NET Framework平台下的VB.NET和VC与之前的版本软件也是存在一些不同的。

在本章中，主要学习了C#语言的一些特点，以及它与其他流行的开发语言的相似之处及区别。通过一系列的对比分析，不仅了解到了.NET Framework平台下的C#语言的特性，还知道了VB.NET、VC++，以及Java语言的特质。

1. C#与VB.NET

VB.NET是基于VB语言的，它是一种可简便快捷地创建.NET应用程序的语言。其实VB.NET与C#的异同不是一两句话能比较清楚的。在.NET开发环境下，VB.NET和C#差别不大。在.NET Framework平台下，VB.NET和C#都生成一样的IL，因此理论上说不存在性能的差异。当然，作为两种不同的语言，它们的不同之处也显而易见。

C#书写的代码比VB.NET短小，但在开发环境中，VB.NET的自动完成功能比C#更完善。VB.NET的代码接近完整的英语，比C#更容易理解代码其中的含义及功能。在VB.NET中，通过Events、Try...Catch和Select...Case等语句可以直观地实现相应的功能，但是面向对象性会显得薄弱一些。在C#中，取消了很多面向过程性语法，取而代之的是全部使用类的封装等技术来实现。另外，VB.NET内置了很多东西，像字符串操作和类型转换。C#缺乏这些内置的支持。

2. C#与C++

C#是Microsoft的一种高级编程语言，是Microsoft自己制定的新的标准，因此从语言的构造上来上它只是采用了C++的少数特性，跟C++具有非常大的不同。

从语言的发展历程来看，C++从C发展而来，C是面向过程的，而C++是面向对象的，两者都是DOS下的编程工具，但VC++完成了C++的可视化，因此可以在Windows平台上方便用户运行。

把现有C/C++代码放到C#中运行是不可能编译通过的。所以，如果需要移植C/C++代码，最好使用Visual C++ .NET。它做到了最好的新旧结合。

3. C#与Java

形象地说,Java站在了C++的肩膀上,而C#则站在了Java的肩膀上。也许正是由于这种“血缘”上的联系,使得C#与Java语言拥有了很多的相同或者相似之处。此外,C#如Java一样,通过捕捉和抛出异常对象来管理错误处理过程。当然,如同在第1章介绍过的一样,C#借鉴了Java的内存管理方式,由底层.NET框架进行自动内存垃圾回收,Java的中间代码和MSIL都是中间的汇编形式的语言,它们在运行时或其他时候被编译成机器代码。

C#与Java在常量、基类、构造函数、静态构造函数、虚函数、ref和out参数、定义类型等方面有着很大的不同。

与Java的各具特色的优越特性相比,C#最大的卖点可能就是它与COM的无缝集成了。COM是Microsoft的Win32组件技术,因此本质上讲Visual Studio .NET的用户可在任何.NET语言里编写COM客户和服务端程序。C#类可以子类化为一个已经存在的COM组件,生成的类也能被作为一个COM组件使用,这些都是Java语言所不具备的。

在实际的开发工作中,这些语言都有着各自的优良特点。但是C#将面向对象、类型安全、组件技术、跨平台异常处理、自动内存管理、代码安全管理和版本控制等诸多要素整合到了一起,形成了一种兼具诸多优良特性的语言。通过性能等多方面的比较之后,读者应更加清楚地意识到,在.NET平台下,C#语言具有一些其他语言不可比拟的优势,这使它成为整个Visual Studio .NET环境中最受用户青睐的一种开发语言。

2.8 实战练习

1. 启动Visual Studio 2010,新建一个控制台应用程序,输入2.2节中的示例程序,编译并运行该程序。
2. 新建一个控制台应用程序,输入2.5.9节中的示例程序,编译并运行程序。
3. 新建一个控制台应用程序,输入2.5.10节中的示例程序,编译并运行程序。

第2篇 C#程序设计基础

- ▶▶ 第3章 C#数据类型
- ▶▶ 第4章 变量与表达式
- ▶▶ 第5章 程序控制语言
- ▶▶ 第6章 函数与方法

第 3 章 C#数据类型

从本章开始，暂时离开 Visual Studio 2010 的 IDE，直接进入代码的学习中。计算机程序处理的对象是数据，而数据是以整数、小数和字符等特定的形式存在的。在 C#语言中，变量或常量就是通过指定特定位置来存放数据的存储单元。通常来说，一个变量或者常量都会对应着一定的数据类型，如整型、字符型和布尔型等。一个以上的某种类型的变量或者常量在特定运算操作符的修饰下就形成了表达式。

在 C#的语法定义中，一般将数据类型划分为内置数据类型，以及用户定义数据类型。数据类型还可以分为值类型和引用类型两种。值类型通常用于存储值，引用类型则用于存储对实际数据的引用。本章主要向读者详细介绍 C#语言数据类型的相关知识。

3.1 初识 C#的数据类型

C#中有两种数据类型，即值类型（value type）和引用类型（reference type）。值类型变量存储的是数据的实际值，而引用类型变量存储的是它们的数据引用。值类型的每一个变量都有自己的数据复制。除了 ref 和 out 参数变量之外，对值类型变量而言，改变一个变量的操作不影响其他变量。而对于引用类型而言，存在两个变量引用相同对象的可能。因此对这两个变量中的任何一个变量的操作都有可能影响另一个对象引用的对象。表 3.1 给出了 C#数据类型的基本信息。

表 3.1 C#数据类型概述

类 别		描 述
值类型	简单类型	有符号整型: sbyte,short,int,long
		无符号整型: byte,ushort,uint,ulong
		Unicode 字符: char
		浮点型: float,double
		高精度小数: decimal
		布尔型: bool
引用类型	枚举类型	自定义类型 enum E{...}
	结构类型	自定义类型 struct S{...}
	类类型	所有其他类型的最终基类: object
		Unicode 字符串: string
		自定义类型 class C{...}
	接口类型	自定义类型 interface I{...}
	数组类型	单维与多维数组，例如，int[]与 int[,]
	委托类型	自定义类型 delegate T D(...)

说明：引用类型可以被认为是对象。

关于 C#数据类型的具体知识,将在本章的各个小节中逐个具体讨论。对于有计算机语言知识基础的读者而言,任何语言的数据类型都有一些相同之处。

- 整型类型,支持有符号或者无符号的 8 位、16 位、32 位和 64 位整数。
- 浮点类型,包括 float 和 double 两种类型, float 型表示 32 位单精度的 IEEE754 格式, double 型表示 64 位双精度的 IEEE754 格式。
- 小数类型, decimal, 表示 128 位十进制小数,应用于货币等方面的计算。
- 布尔类型, bool, 表示 true 或者 false 类型的布尔值。
- 字符类型, char, 表示 16 位的 Unicode 编码单元。
- 字符串类型, string, 表示 16 位的 Unicode 编码单元的序列。

在 C#中,字符和字符串的处理使用 Unicode 编码。


在 C#中,简单类型是系统预先提供的一套定义好的结构类型。简单类型中包含很多用来定义的保留字,这些保留字都是在 System 命名空间(namespace)里预定义好的结构类型。例如, int 是简单类型中一个 32 位整型保留字,而 System.Int32 是在 System 命名空间中的一个预定义类型。一个简单类型同其化名的结构类型是完全一样的。也就是说, int 和 System.Int32 是一样的意义。从这个意义上讲,数据类型的划分也可以如下:

- 简单类型主要有整型、浮点类型、小数类型、布尔类型和字符型。
- 值类型包括简单类型、集合类型和结构类型。
- 引用类型包括类类型、接口类型、代表类型和数组类型。

在本书中,主要考虑值类型与引用类型这两种类型的划分。

值类型和引用类型的不同之处在于:值类型的变量值直接包含数据,而引用类型的变量则将其引用存储在对象中。值类型在堆栈(stack)上分配,这样就可以确保.NET 平台下的大多数程序语言都能应用它。而引用类型则被分配在堆(heap)上,这种类型变量通常作为类的实例的代表。值类型在内存中引用的时候会在堆栈(stack)中创建一个全新的副本,而不是简单的引用。而引用类型仅仅是对对象地址的简单引用。在 C#代码中,用户可以根据需要来定义值类型或者引用类型。

在编写程序时,完全可以让两个不同的引用类型的变量引用同一个对象。这种情况下对这两个变量中任何一个变量的操作都可能会影响到另一个变量引用的对象。而每一个值类型变量有其自己的数值,因此对其中一个变量的操作不可能影响到另外一个变量。

 **说明:** 所有值类型以及引用类型都由对象类(object)发展而来。在 C#中可通过隐性转换或显性转换来改变数据类型。隐性转换常常被应用于不同类型的变量之间的转化,这种方式一般不会造成数据丢失;显性转换通常被应用于拆箱(unboxing)操作,有可能会造成数据丢失或者降低数据的精确度。

3.2 存储实际数据的值类型

3.2.1 什么是值类型

在 C#中,所有被预定义的值类型都隐含地声明了一个公共的无参数的默认构造函数。每个默认构造函数都会返回一个初始为 0 的值类型的实例,这些实例其实就是为大家所熟知的默认值。如表 3.2 给出了 C#语言中所有值类型的默认值。

表 3.2 数据类型表

C#数据类型	大 小	默 认 值
uint	无符号的 32 位整数	0
int	有符号的 32 位整数	0
float	32 位浮点数, 精确到小数点后 7 位	0.0F
double	64 位浮点数, 精确到小数点后 15~20 位	0.0D
decimal	128 位浮点数, 精确到小数点后 28~29 位	0.0M
sbyte	有符号的 8 位整数	0
byte	无符号的 8 位整数	0
ushort	无符号的 16 位整数	0
short	有符号的 16 位整数	0
ulong	无符号的 64 位整数	0
long	有符号的 64 位整数	0L
bool	布尔值, true 或 false	false
string	Unicode 字符串	-
char	单个 Unicode 字符	'\x0000'


这里需要补充说明以下几种特殊类型的默认值:

- ☐ 枚举类型的默认值是 0;
- ☐ 结构类型的默认值是根据类型的不同而有所变化;
- ☐ 值类型都设置为各自的默认值;
- ☐ 引用类型默认赋值为空。

3.2.2 整型

C#中用整型变量来存储整数。根据变量所能表示数据的范围不同, 又将整型划分为字节型 (byte 和 sbyte)、短整型 (short 和 ushort)、整型 (int 和 uint)、长整型 (long 和 ulong) 等 8 种类型。

有符号整型与无符号整型在定义与语法上没有太多差异, 各种无符号类型变量所占的内存空间字节数与相应的有符号类型量相同。但由于省去了符号位, 所以无符号类型变量不能表示负数。然而他们使用场合却稍有不同。当进行编译时, 会根据是否有符号来决定具体的机器代码指令。而有无符号类型的值或者变量, 则是为了配合编译器的基本原理与性能才发挥作用的。实际上, 有符号和无符号数据在进行不同的运算方式时必须需要机器代码支持, 否则有无符号数除了增加机器指令之外再无其他意义。

 **说明:** 在操作系统的基础代码及接口中, Win32 的 API 中广泛使用了无符号整型。在函数定义等数据结构中, 16 位的无符号整型和 32 位无符号整型发挥着重要的作用。将无符号整型强制转换到有符号整型, 可以在从 Win32 类型传递到一个 CLI (Command - Line Interface, 一种对 IOS 操作系统的设置方式) 兼容类型时被应用。这时用户就可能需要使用 C#的 unchecked 关键字来避免溢出错误。

3.2.3 字符型

首先需要强调一下 `char` 类型与其他整数类型的两点不同之处。

1. 不能进行其他类型到`char`类型的隐式转换

尽管 `char` 类型可以完全代表 `sbyte`、`byte` 和 `ushort` 等类型,但是 C# 仍然不允许将 `sbyte`、`byte` 和 `ushort` 类型隐式转换到 `char` 类型。

2. 只能以字符形式表示`char`类型的常量

在 C# 中,这是必须遵循的原则。如果用整数形式表示 `char` 类型的常量,则必须带有类型转换前缀。

例如: `(char) 10`。

既然没有隐式转换,程序员要如何处理 `char` 这种特殊的值类型呢?一般情况下,可以通过赋值操作来完成这种转换。

赋值形式有以下 3 种:

- ☐ `char Ach="A";`
- ☐ `char Ach="\x0065";` //十六进制;
- ☐ `char Ach="\u0065";` //unicode 表示法。

通过以上的例子,相信读者已经掌握了字符型变量赋值操作的方法。在第 2、3 行示例代码中,出现了一些“奇怪的”代码。

“`\x0065`”以及“`\u0065`”到底代表什么呢?其实,除了赋值操作以外,在编写程序的过程中,`char` 型还要经常与转义符打交道。在 C# 中,转义符 (Escape Sequences) 是一种表示字符的特殊形式,它是从 C 中直接借鉴并继承而来的,用来表示 ASCII 码字符集中某些特定功能的字符,以及不可打印的控制字符。

字符型转义符如表 3.3 所示:

表 3.3 转义符列表

转义符名称	转义符型式	ASCII 码值 (十进制)	Unicode 编码 (十六进制)
单引号	<code>\'</code>	039	0x0027
双引号	<code>\"</code>	034	0x0022
感叹号 (警报)	<code>\a</code>	007	0x0007
反斜杠	<code>\\</code>	092	0x005C
空字符	<code>\0</code>	000	0x0000
退格符	<code>\b</code>	008	0x0008
回车符	<code>\r</code>	013	0x000D
换行符	<code>\n</code>	010	0x000A
换页符	<code>\f</code>	012	0x000C
水平 tab	<code>\t</code>	009	0x0009
垂直 tab	<code>\v</code>	011	0x000B
1~3 位八进制的任意字符	<code>\ddd</code>		
1~2 位十六进制的任意字符	<code>\xhh</code>		

3.2.4 浮点型

下面再来介绍一下浮点类型。C#语言中提供了两种浮点类型：单精度（float）类型和双精度（double）类型。

- float 型，能表示的值的范围为 $1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$ ，精确到小数点后面 7 位。
- double 型，能表示的值的范围为 $5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$ ，精确到小数点后面 15 位或 16 位。

当二元表达式中有一个操作数为浮点类型，另外一个操作数是整型或浮点类型时，那么运算规则如下：

- 如果另一个操作数是整型，则被自动转换为浮点数类型；
- 如果操作数之一为 double，则另一操作数经过相应的转换也变成 double 型，那么运算以 double 类型的精度和取值范围进行，并且所得结果也为 double 类型；
- 如果以上两种情况都不满足时，运算至少以 float 类型的取值范围和精度进行，并且所得结果也为 float 型。

下面一段示例代码主要为读者演示了 float 类型与 double 类型在运算中精度的取舍问题。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._2._4
{
    //定义 Program 类
    class Program
    {
        //定义程序的 Main() 方法
        static void Main(string[] args)
        {
            //定义变量，并进行初始化
            int x = 10;
            float y = 4.5f;
            short z = 6;
            double w = 2.3E+3;
            //输出结果的类型为 double:
            Console.WriteLine("The sum is {0}", x + y + z + w);
            Console.Read();
        }
    }
}
```

在示例中，4 个变量赋值后，一个 int 类型、一个 short 类型、一个 float 类型和一个 double 类型相加，计算结果为 double 类型。输出结果为：

```
The sum is 2324.5
```



3.2.5 小数型

在 C# 语言中，有一种新引进的数据类型 `decimal`。小数类型非常适用于金融和货币运算。数值范围为 $1.0 \times 10^{-28} \sim 7.9 \times 10^{28}$ ，精确到小数点后面 28 位。

在二元操作中，小数型运算的操作法则如下：

- ❑ 当一个操作数是小数类型，另一个操作数是整型或者小数类型时，整型操作数在运算前被编译器自动转化为小数类型再进行运算。
- ❑ 当小数类型的算术运算产生大于该格式的值时，系统会触发溢出错误。
- ❑ 当小数类型的算术运算产生了小于该格式的值时，操作的结果将会变成 0。

很明显，小数类型的精确度要远远大于浮点类型，但是其数值范围却要比浮点类型小了很多。对于一般的变量或者常量而言，浮点类型转化为小数类型时会产生溢出错误，而小数类型转化为浮点类型时会造成精确度的损失。

 注意：小数型和浮点型这两种类型之间不存在隐式或显式转换。

3.2.6 大整数型

在 C# 4 中，新引进一种能表示任意大的带符号整数的类型 `BigInteger`，定义在 `System.Numerics` 命令空间中。`BigInteger` 类型是不可变类型，代表一个任意大的整数，其值在理论上已没有上部或下部的界限。`BigInteger` 类型的成员与其他整数类型的成员近乎相同，不同的是，`BigInteger` 类型没有 `MinValue` 和 `MaxValue` 属性，也就是说它没有没有上限或下限，对于导致 `BigInteger` 值增长过大的任何操作会引发 `OutOfMemoryException`。

与普通整数类型不同，对 `BigInteger` 对象可通过以下多种方式实例化：

- ❑ 可以声明 `BigInteger` 变量并向其分配一个值，分配的值可以是您需要的任何数值，只要该值为整型即可。
- ❑ 可以使用 `new` 关键字并提供任何整数或浮点值以作为 `BigInteger` 构造函数的一个参数。
- ❑ 可以使用 `new` 关键字并向 `BigInteger.BigInteger` 构造函数提供任意大小的字节数组。此构造函数仅用于创建正的 `BigInteger` 值。
- ❑ 可以调用 `BigInteger` 方法，对数值表达式执行某些操作并返回计算的 `BigInteger` 结果。

例如，以下代码都可以初始化一个 `BigInteger`：

```
namespace _3._2._6
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger bigIntFromDouble = new BigInteger(179032.6541);
            BigInteger assignedFromLong = 6315489358112;
            BigInteger newBigInt = new BigInteger(
                new byte[] { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 });
        }
    }
}
```



```

        Console.WriteLine("bigIntFromDouble:{0}", bigIntFromDouble);
        Console.WriteLine("assignedFromLong:{0}", assignedFromLong);
        Console.WriteLine("newBigInt:{0}", newBigInt);

        Console.ReadLine();
    }
}

```

程序运行结果如下：

```

bigIntFromDouble:179032
assignedFromLong:6315489358112
newBigInt:759477275222530853130

```

注意最后一个通过数组实例化的 `new BigInt` 的输出结果，对于通过字节数组初始化时，数组中的每个字节的内容将被认为是一个十六进制的数，因此 `new BigInt` 的值为 `0x102030405060708090a`（按数组中下标由高到低），输出为十进制时就得到上面的结果了。

可以像使用其他任何整数类型一样使用 `BigInteger` 实例。`BigInteger` 重载标准数值运算符，能够执行基本数学运算，如加法、减法、除法、乘法、减法、求反和一元求反。还可以使用标准数值运算符对两个 `BigInteger` 值进行比较。与其他该整型类型相似，`BigInteger` 还支持按位 `And`、`Or`、`XOr`、左移位和右移位运算符。对于不支持自定义运算符的语言，`BigInteger` 结构还提供了用于执行数学运算的等效方法。其中包括 `Add`、`Divide`、`Multiply`、`Negate`、`Subtract` 和多种其他内容。此外，`BigInteger` 增加了成员，例如：

- ☐ `Sign`，可以返回表示 `BigInteger` 值符号的值。
- ☐ `Abs`，可以返回 `BigInteger` 值的绝对值。
- ☐ `DivRem`，可以返回除法运算的商和余数。
- ☐ `GreatestCommonDivisor`，可以返回两个 `BigInteger` 值的最大公约数。

3.2.7 复数型

在 `System.Numerics` 命令空间中还提供一个处理复数的类型 `Complex`。

复数是由实数部分和虚数部分组成的数。复数 z 通常的书写形式为 $z = x + yi$ ，其中 x 和 y 为实数， i 为虚数单位，具有 $i^2 = -1$ 的特性。复数的实数部分由 x 表示，复数的虚数部分由 y 表示。

在实例化和操作复数时，`Complex` 类型使用笛卡尔坐标系统（实数，虚数）。一个复数可以表示复杂平面的二维坐标系中的某个点。复数的实数部分位于 x 轴（水平轴），虚数部分位于 y 轴（垂直轴）。

通过使用极坐标系，复杂平面中的任何点还可以基于其绝对值表示。在极坐标中，一个点表示为两个数字：

- ☐ 它的量值，是指从该点到原点（即 $0,0$ 或 x 轴和 y 轴线的交点）的距离。
- ☐ 它的相位，是指真实的轴与从原点到该点绘制的线条之间的角度。

可以通过以下方法之一为复数赋值：

- ❑ 通过将两个 `Double` 值传递给它的构造函数。第一个值表示复数的实数部分，第二个值表示其虚数部分。这些值表示二维笛卡尔坐标系统中复数的位置。
- ❑ 通过调用静态 `Complex.FromPolarCoordinates` 方法从点的极坐标创建复数。
- ❑ 通过将 `Byte`、`SByte`、`Int16`、`UInt16`、`Int32`、`UInt32`、`Int64`、`UInt64`、`Single` 或 `Double` 值赋值给 `Complex` 对象。该值将成为复数的实数部分，其虚数部分等于 0。
- ❑ 通过将 `Decimal` 或 `BigInteger` 值强制转换为 `Complex` 对象。该值将成为复数的实数部分，其虚数部分等于 0。
- ❑ 通过将方法或运算符返回的复数赋值给 `Complex` 对象。例如，`Complex.Add` 是一个静态方法，它返回一个属于两个复数之和的复数，`Complex.Addition` 运算符会计算两个复数之和，然后返回结果。

例如，以下代码就可完成复数的实例化和运算。

```
namespace _3._2._7
{
    class Program
    {
        static void Main(string[] args)
        {
            Complex z1 = new Complex(12, 6);
            Complex z2 = 3.14;

            Complex r1 = Complex.Add(z1, z2);
            Complex r2 = Complex.Subtract(z1, z2);

            Console.WriteLine("z1+z2={0}", r1);
            Console.WriteLine("z1-z2={0}", r2);

            Console.ReadLine();
        }
    }
}
```

3.2.8 布尔型

在 C# 中，布尔类型是专门用来表示“真”和“假”两个概念。布尔类型变量只有 `true` 和 `false` 两种取值。`true` 代表“真”，`false` 代表“假”。没有任何方法或者标准可以实现布尔类型与其他类型的转换。

3.2.9 C#值类型的数值类型

经过以上的分析，下面总结一下 C#值类型的范围与精度。如表 3.4 所示，读者可以直观地了解到所有 C#中值类型的取值原则（不包含 `BigInteger` 类型）。了解了取值的原则不仅可以在以后的程序设计与编码过程中避免发生溢出错误，而且可以提高代码的健壮性和安全性。

表 3.4 C#的数值类型

类 别	位 数	类 型	范围 / 精度
有符号整型	8	sbyte	-128~127
	16	short	-32 768~32 767
	32	int	-2 147 483 648~2 147 483 647
	64	long	-9 223 372 036 854 775 808~9 223 372 036 854 775 807
无符号整型	8	byte	0~255
	16	ushort	0~65535
	32	uint	0~4 294 967 295
	64	ulong	0~18 446 744 073 709 551 615
浮点型	32	float	$1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$, 精度 7 位
	64	double	$5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$, 精度 15 位
Decimal	128	decimal	$1.0 \times 10^{-28} \sim 7.9 \times 10^{28}$, 精度 28 位

在介绍完 C# 值类型的基本种类之后, 这里要补充一点关于值类型的细节。实际上, 值类型可以进一步划分为简单类型 (simple type)、枚举类型 (enum type) 和结构类型 (struct type)。以上所述的几种类型都只是简单类型的组成部分, 而关于枚举类型与结构类型将留在后面的章节中单独讨论。

3.3 存储引用地址的引用类型

3.3.1 什么是引用类型

顾名思义, 引用类型的重点就是“引用”二字。这里的“引用”是指该类型的变量是指向其所要存储的地址单元, 而不直接存储包含的值本身。引用类型存储实际数据的引用值的地址, 如图 3.1 所示。

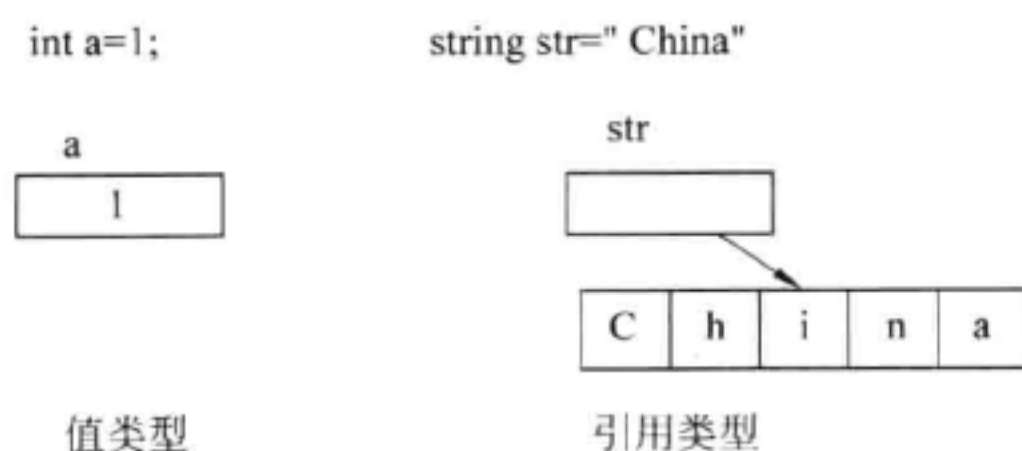



图 3.1 值类型与引用类型

引用类型主要包括类类型、对象类型、字符串类型、接口类型、代表类型和数组类型。本节先详细介绍一下前 3 种引用类型。

3.3.2 类类型

类类型是引用类型中的一种数据结构, 它包含了如常量、字段和事件等的数据成员,

还有如方法、属性、索引、操作、构造函数和析构函数等的函数成员，以及嵌套类型。类支持继承和多态，这些机制确保派生类能够扩展和特殊化基类。

 **注意：**和 C++ 相比，C# 中的类仅允许单继承，派生一个新对象不能有多重基类，但允许一个类派生自多重接口。

3.3.3 对象类型

在前面已经提及，在 C# 中，其他所有类型最终的基础类型是对象类型。在 C# 中，每一种类型都直接或者间接的源于 `object` 这个类类型。对象类型变量是用关键字 `object` 来定义的，它是 `System.Object` 的简化别名。因为对象类型是所有对象的基类，所以可把任何类型的值赋给它。


定义对象类型变量的一般格式：

```
object 变量名;
```

在程序中经常能看到用 `object` 定义类型的方法的参数。

3.3.4 字符串类型

字符串类型是一个密封类，本质上而言它直接从 `object` 中继承而来。同其他类型一样，由于单继承性的限制 C# 不允许从字符串类型再派生类。字符串类型变量用关键字 `string` 来定义，它是 `System.String` 的简化别名。在程序设计语言中，字符串类型是一个最基本的数据类型。C# 中规定 `String`（或 `string`）类的对象在创建后就不能再对其进行修改。可以用字符串文字的形式来表示 `string` 类型的值。

 **注意：**没有任何操作能够改变 `string` 类对象的状态。

定义字符串变量的一般格式为：

```
string 变量名;
```

例如：

```
string str="Made'n";
```

C# 中可以使用 “+” 连接字符串。例如：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._3._4
{
    //定义 Program 类
    class Program
    {
        //定义类的私有成员变量
```

```

static string str1 = "Made in ";
static string str2 = "China";
//定义程序的Main( )方法
static void Main(string[] args)
{
    //打印输出结果
    Console.WriteLine(str1 + str2);
    Console.ReadLine();
}
}

```

输出结果如下：

```
Made in China
```

我们还可以比较两个字符串是否相等。比较两个字符串是否相等时，简单地使用“==”比较操作符即可。例如：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._3._4
{
    //定义 Program 类
    class Program
    {
        //定义类的私有变量，并进行初始化
        static string str1 = "Made in";
        static string str2 = " China";
        //定义类的Main( )方法
        static void Main(string[] args)
        {
            //方法的处理过程
            if (str1 ==str2)
            {
                Console.WriteLine("两个字符串相同");
                Console.ReadLine();
            }
            else
            {
                Console.WriteLine("两个字符串不同");
                Console.ReadLine();
            }
        }
    }
}

```

输出结果为：

```
两个字符串不同
```

另外，如果用户想访问某个字符串中的单个字符时，可以通过访问字符串下标的方式实现。接着上面的实例程序，可以这样编写代码：

```

//声明使用的命名空间
using System;

```



```

using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._3._4
{
    //定义 Program 类
    class Program
    {
        static string str1 = "Made in China";
        //访问字符串 str1 的第一个字符
        static char ch1 = str1[0];

        static void Main(string[] args)
        {
            //打印输出结果
            Console.WriteLine("字符串 str1 是{0}", str1);
            Console.WriteLine("字符串 str1 的第一个字符是{0}", ch1);
            Console.ReadLine();
        }
    }
}

```

程序输出结果是：

```


字符串 str1 是 Made'n China
字符串 str1 的第一个字符是 M

```

3.3.5 接口类型

接口类型定义一组仅存在方法标志，但没有执行代码的函数成员命名的集合。在 C# 中，编译器不允许实例化一个接口，用户只能实例化一个派生自该接口的对象。类或结构能实现多个接口。一个接口可以声明一个只有抽象成员的引用类型。

实际上，由于 C# 中不允许多根继承，所以用于实现接口的类或结构就担负起了实现接口函数成员的责任，也就是实质意义上的多继承。当程序员定义从接口派生的类时，这个类可以派生自多重接口。而当实现类从类的派生时，C# 就只允许从一个类派生，即单根继承。

 **说明：**接口可能从多个基接口继承而来，可以在一个接口中定义方法、属性和索引。

接口的声明方法如下例：

```

//声明 Iface 接口
interface Iface
{
    void showmyIface();
}

```

类和接口最重要的区别在于，类可以实例化并且被实现，而接口不能被实现。接口是面向对象编程技术的基石之一，正是由于有了接口才使得程序操作对象称为一种十分灵活的方式。

由于接口部分涉及太多面向对象的概念，在背景知识不是很齐备的情况下，此处先不

做具体介绍了。感兴趣的读者可以参读下面的示例程序，体会一下接口定义的意义所在。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._3._5
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Write text = new Write("OK");
            IPoint iPoint = (IPoint)text;
            iPoint.PointX(0.123D);
            iPoint.PointY(0.234D);
            iPoint.PointZ(0.345D);
        }
    }
    public class Draw
    {
        public Draw() { }
    }
    //声明接口
    interface IPoint
    {
        void PointX(double factor);
        void PointY(double factor);
        void PointZ(double factor);
    }
    //定义 Write 类，继承自接口 IPoint 以及类 Draw
    public class Write : Draw, IPoint
    {
        //定义 Write 类的成员方法
        public Write(string text)
        {
            this.text = text;
        }
        public void PointX(double factor)
        {
        }
        public void PointY(double factor)
        {
        }
        public void PointZ(double factor)
        {
        }
        //私有成员变量
        private string text;
    }
}
```

3.3.6 dynamic 类型

dynamic 类型是 C# 4 中新增加的数据类型。dynamic 类型在编译期间忽略类型检查，

给 `dynamic` 类型的对象定义任何操作编译器都认为是有效的，也就是说，在代码运行之前是不会检查错误的。

其实，在 .NET Framework 以前的版本中通过 `var` 关键字和匿名方法也为 C# 提供了类似“动态编程”的方法，只是 `var` 类型的对象在运行过程中不允许再更改类型，而 `dynamic` 类型却可以。例如，有以下代码：

```
namespace _3._3._6
{
    class Program
    {
        static void Main(string[] args)
        {
            var varTest = 1;
            dynamic dynTest = 1;

            Console.WriteLine("varTest:{0}", varTest.GetType());
            Console.WriteLine("dynTest:{0}", dynTest.GetType());

            //varTest = "Hello";    //若不注释该行，程序不能被编译
            dynTest = "Hello";

            Console.WriteLine("varTest:{0}", varTest.GetType());
            Console.WriteLine("dynTest:{0}", dynTest.GetType());

            Console.ReadLine();
        }
    }
}
```

程序输出结果是：

```
varTest:System.Int32
dynTest:System.Int32
varTest:System.Int32
dynTest:System.String
```

从以上运行结果可看到，`dynTest` 最初的类型是 `Int32`，当被赋值一个字符串后，其数据类型又变为了 `String`。

以上程序中，分别定义了一个 `var` 类型和 `dynamic` 类型的变量，若去掉程序中注释掉的那行语句，编译程序时将得到如下所示错误：

无法将类型“string”隐式转换为“int”

而对于 `dynamic` 类型，则可以在程序运行期间更改其数据类型，如上例中最初 `dynTest` 变量是整型，后来又将一个字符串赋值给该变量，这样，该变量就会改变为字符串类型。

从上例可看到，`dynamic` 类型的使用非常灵活。在使用时需要注意以下两个方面的问题：

- ❑ 当变量定义为 `dynamic` 类型时，由于在设计期间其具体类型无法确定，因此无法使用 Visual Studio 的 IntelliSense 进行成员提示了。
- ❑ 由于编译器将忽略 `dynamic` 类型变量的检查，因此必须确保调用的属性或方法是正确的，否则，在程序运行时将出现异常（在编译期间无法检查出这些异常）。

在 C# 4 中提供 `dynamic` 支持的是称为 Dynamic Language Runtime（简称 DLR）的一系

列类库。DLR 是构建于 CLR 之上的，用来提供动态语言支持的类库。

3.4 数据类型是可以转换的

3.4.1 什么时候发生类型转换

在程序中，值的复制主要是指赋值运算，以及函数方法传递参数。而类型转换主要发生于被赋值的变量（或方法的形式参数）的类型与实际的对象类型不一致的情况。当发生类型转换时，源类型（Source Type）是指实际对象的类型，目标类型（Destination Type）即被赋值的变量（或方法的形参）的类型。

C#中的类型转换有两种分类方法，一种是根据转换方式的不同可以分为显式（Explicit）转换和隐式（Implicit）转换两种；另外一种是根据源类型和目标类型之间的关系分为投射（Cast）、变换（Conversion）和封箱/拆箱（Boxing/Unboxing）。

从直观上看，显式和隐式转换只是语法上存在一些不同，其他差别不大。下面将展开深入的讨论来分析二者之间到底有何不同。

C#表示一个显式转换的语法构造形如“（目标类型）”。

下面的代码介绍了在 C#语言中进行显式和隐式转换时的语法。

```
int x = 10;
//隐式转换
long y = x;
//显示转换
x = (int) y;
```

3.4.2 显式转换

下面来重点谈一谈显式转换。当发生类型转换时，假如在代码中明确指定了目标类型，则称为显式转换，否则就称为隐式转换。表 3.5 列举了所有 .NET Framework 中各个数据类型之间能够发生显式转换的类型名称及转化关系。

表 3.5 显式转换

源 类 型	目 标 类 型
Byte	sbyte 或 char
Sbyte	Byte、ushort、uint、ulong 或 char
Int	sbyte、byte、short、ushort、uint、ulong 或 char
UInt	sbyte、byte、short、ushort、int 或 char
Short	sbyte、byte、ushort、uint、ulong 或 char
Ushort	sbyte、byte、short 或 char
Long	sbyte、byte、short、ushort、int、uint、ulong 或 char
Ulong	sbyte、byte、short、ushort、int、uint、long 或 char
Float	sbyte、byte、short、ushort、int、uint、long、ulong、char 或 decimal

续表

源 类 型	目 标 类 型
Double	sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 decimal
Char	sbyte、byte 或 short
Decimal	sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 double

3.4.3 隐式转换

接着 3.4.2 节的介绍, 这里重点介绍隐式转换的方向, 以及类型范围。表 3.6 列举了所有 .NET Framework 数据类型之间的隐式转换。

表 3.6 隐式转换

源 类 型	目 标 类 型
Byte	short、ushort、int、uint、long、ulong、float、double 或 decimal
Sbyte	short、int、long、float、double 或 decimal
Int	long、float、double 或 decimal
UInt	long、ulong、float、double 或 decimal
Short	int、long、float、double 或 decimal
Ushort	int、uint、long、ulong、float、double 或 decimal
Long	float、double 或 decimal
Ulong	float、double 或 decimal
Float	double
Char	ushort、int、uint、long、ulong、float、double 或 decimal

3.4.4 不同数值类型之间的转换

在 3.2.1 节中已经讲过, 有符号的数值类型主要包括 byte、short、int、long、float、double 等。依照这个排列顺序向后排列, 编译器允许各种类型的值依次自动进行转换为其后的数值类型。这就是说, 当一个 short 型变量赋值给一个 double 型的变量时, 编译器首先做出判断, 确定允许隐式转换以后, short 型变量的值将会被自动进行转换成 double 型的值, 然后再将已经转换后的值赋给 double 型变量。

在下面的示例程序中, 将具体演示两种数值类型之间的转换。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._4._4
{
    //定义 Program 类
    class Program
    {
        //定义 Program 类的静态成员变量
        static byte a = 1;
    }
}
```

```

static short b = a;
static int c = a;
static long d = a;
static float e = a;
static double f = a;
static void Main(string[] args)
{
    //打印输出结果
    Console.WriteLine("byte a = " + a);
    Console.WriteLine("short b = " + b);
    Console.WriteLine("int c = " + c);
    Console.WriteLine("long d = " + d);
    Console.WriteLine("float e = " + e);
    Console.WriteLine("double f = " + f);
    Console.Read();
}
}

```

由于初始化的 byte 型变量是数值类型序列中最靠前的,所以它可以被自动转化为其余任何一种类型的变量。这个过程其实就是一个隐式转换的过程。

程序的输出结果如下:


```

byte a =1
short b=1
int c =1
long d =1
float e =1
double f =1

```

当 int 型转换为 short 型时,则使用强制类型转换。强制转换的格式如下:

(类型名) 变量名

 **注意:** 使用强制转换必须要考虑不同类型数值范围的问题。进行转换时,被转换的数据大小一定要控制在目标类型的范围之内。如果将 byte 的 334 转换为 sbyte,系统就会溢出。所以,在多字节数据类型转换为少字节类型的时候,必须考虑到转换的范围问题。此外,即使字节数相同的有符号类型和无符号类型之间转换,也要考虑转换范围等条件。

3.4.5 数值类型和字符串之间的转换

在 C# 中,用一对双引号包含的若干字符来表示字符串,如“123”,这既是一串字符,也是一个数。实际上在 C# 的语法中,“123”其实是数值字符串。因此在实际的编码工作中,当需要将数字转化为字符串时,可以调用 void ToString() 方法。C# 中的所有类都具有这个方法,这大大简化了数值转换成字符串的过程。

反过来,将数值型字符串转换成数值该怎么处理呢?在 C# 语言中,short、int、float 等数值类型均有一个静态函数 Parse()。这个函数就是用来将字符串转换为相应数值的。下面的例子可以更明确地说明转换的方法:

```
//声明使用的命名空间
```



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._4._5
{
    //定义 Program 类
    class Program
    {
        static float f = 3.21F;
        static string str = "123";
        static void Main(string[] args)
        {
            Console.WriteLine("f = " + f.ToString());
            //定义 Main( ) 方法的处理过程
            if (int.Parse(str) == 123)
            {
                Console.WriteLine("string str =" + str);
                Console.WriteLine("int.Parse(str) = " + int.Parse(str));
                Console.WriteLine("转换成功 ");
                Console.Read();
            }
            else
            {
                Console.WriteLine("转换失败 ");
                Console.Read();
            }
        }
    }
}

```


运行结果如下:

```

f = 3.21
string str =123
int.Parse(str) =123
转换成功

```

在上面提到了 ToString() 方法可以将数值转换成字符串, 不过在字符串中, 结果是以十进制显示的。在转换过程中, 数据的存储只与输入输出有关, 与进制没有关系。现在可以使用 ToString(string) 方法带给它加一些参数, 就可以将其转换成十六进制。

 **说明:** 这里需要一个 string 类型的参数, 这就是格式说明符。

1. 八进制

其格式说明符是 “o” 或 “O”。任何一个八进制的数值都必须在前面加上这个格式说明符。例如:

```

int a = 100;    //十进制
int a = 0144;   //转换成八进制表示

```

2. 十六进制

其格式说明符是 “x” 或 “X”, 使用这两种格式说明符的区别主要在于 A~F 六个数

字：“x”代表a~f使用小写字母表示，而“X”表示A~F使用大写字母表示。代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._4._5
{
    //定义 Program 类
    class Program
    {
        static int a = 188;
        static void Main(string[] args)
        {
            //打印输出结果
            Console.WriteLine("a(10) = " + a.ToString());
            Console.WriteLine("a(16) = " + a.ToString("x"));
            Console.WriteLine("a(16) = " + a.ToString("X"));
            Console.Read();
        }
    }
}
```

运行结果如下：

```
a(10) = 188
a(16) = bc
a(16) = BC
```

为了显示结果的整齐，需要控制十六进制表示的长度，如果长度不够，用前导的0填补。我们只需要在格式说明符“x”或者“X”后写上表示长度的数字就解决这个问题了。比如，要限制在4个字符的长度，可以写成“X4”。在上例中追加一句：

```
this.textBox1.AppendText("a(16) = " + a.ToString("X4") + "\n");
```

其结果将输出：

```
a(16)=00BC
```

如何将一个表示十六进制数的字符串转换成整型呢？这一转换同样需要借助于Parse()方法，具体说就是Parse(string, System.Globalization.NumberStyles)方法。此方法的第1个参数是表示十六进制数的字符串，如“AB”、“20”（表示十进制的32）等；第2个参数System.Globalization.NumberStyles是一个枚举类型，用来表示十六进制的枚举值是HexNumber。

如果要将“AB”转换成整型代码如下：


```
int b = int.Parse("AB", System.Globalization.NumberStyles.HexNumber);
```

最后得到的b的值是171。

3.4.6 字符的ASCII码和Unicode码之间的转换

在实际工作中，程序员需要得到一个英文字符的ASCII码或者一个汉字字符的Unicode

码时，只要将字符型数据强制转换成适当的数值型数据就可以了。反之，将一个适当的数值型数据强制转换成字符型数据，也可以得到相应的字符。

 **注意：**C#语言中字符的范围不仅包含了单字节字符，也可以包含如中文字符的双字节字符。


在下面的示例代码中，将演示字符的 ASCII 码和 Unicode 码之间的转换。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._4._6
{
    //定义 Program 类
    class Program
    {
        //私有成员变量
        static char ch = 'z';
        static short si = 65;
        static char cn = '人';
        static short uc = 23456;
        static void Main(string[] args)
        {
            //打印转换结果
            Console.WriteLine("The ASCII code of '\" + ch + "\" is: " +
                (short)ch);
            Console.WriteLine("ASCII is " + si.ToString() + ", the char is: " +
                (char)si);
            Console.WriteLine("The Unicode of '\" + cn + "\" is: " + (short)cn);
            Console.WriteLine("Unicode is " + uc.ToString() + ", the char is: " +
                (char)uc + "\n");
            Console.Read();
        }
    }
}
```

运行结果如下：

```
The ASCII code of 'z' is: 122
ASCII is 65, the char is: A
The Unicode of '人' is: 20154
Unicode is 23456, the char is: 宠
```

从这个例子中可以了解到，通过强制转换，完全能够实现字符与字符编码之间的转换。如果程序员需要的不是 short 型的编码，请参考前面介绍的方法进行转换，即可得到 int 等类型的编码值。

 **注意：**当程序员需要从相关的字符编码中查询它是哪一个字符的编码时，也需要用到这种转换。

3.4.7 字符串和字符数组之间的转换


将字符串转化为字符数组时，可以应用字符串类 `System.String` 提供的一个 `void ToCharArray()` 方法实现字符串到字符数组的转换。示例程序如下所示。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._4._7
{
    //定义 Program 类
    class Program
    {
        //私有成员变量
        static string str = "Welcome to Beijing";
        static char[] chars = str.ToCharArray();
        //定义程序的 Main( ) 方法
        static void Main(string[] args)
        {
            //打印输出结果
            Console.WriteLine("string str = " + str);
            Console.WriteLine("Length of \"string\" is " + str.Length);
            Console.WriteLine("Length of \"char array\" is " + chars.Length);
            Console.WriteLine("char[3] = " + chars[3]);
            Console.Read();
        }
    }
}
```

编译程序并运行，运行结果如下：

```
string str = Welcome to Beijing
Length of "string" is 18
Length of "char array" is 18
char[3] = c
```

反之，可以使用 `System.String` 类的构造函数把字符数组转换成字符串。`String(char[])` 和 `String([char[], int, int])` 是 `System.String` 类里两个构造函数，通过观察很容易就能发现，这两个构造函数都携带有字符数组参数。

 **说明：** `String(char[])` 是用一个完备的字符数组的全部元素来构造字符串的；而 `String([char[], int, int])` 则是通过指定字符数组中具体部分来构造字符串的。

下面以 `String(char[])` 为例，说明如何将字符数组转换为字符串。示例程序如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._4._7
```




```

{
    //定义 Program 类
    class Program
    {
        //私有成员变量
        static char[] chars = { 'W', 'e', 'l', 'c', 'o', 'm', 'e' };
        static string str = new String(chars);
        //定义程序的 Main() 方法
        static void Main(string[] args)
        {
            Console.WriteLine("str = \"" + str + "\"");
            Console.Read();
        }
    }
}

```

运行结果为：

```
str = "Welcome"
```

 **注意：**很多情况下需要把字符串转换成字符数组时，只是为了得到该字符串中的某个字符，那么只需要使用 `System.String` 的 `[]` 运算符即可。

3.4.8 字符串和字节数组之间的转换

在 `System.String` 类中没有能实现字符串和字节数组之间的转换方法。要实现这类转换就必须借助 `System.Text.Encoding` 类。在该类中，利用以下方法可以实现字符串与字节数组之间的转换：

- ❑ `byte[] GetBytes(string)` 方法，将字符串转换成字节数组；
- ❑ `string GetString(byte[])` 方法，将字节数组转换成字符串。

`System.Text.Encoding` 类不能通过构造函数来实现编码之间的默认转换，完成这项工作的是默认的 `Encoding` 类的各种属性方法。具体如表 3.7 所示。

表 3.7 `Encoding` 类的各种属性方法

Encoding 属性	作 用
<code>Encoding.Default</code>	获取系统的当前 ANSI 代码页的编码
<code>Encoding.UTF7</code>	获取 UTF-7 格式的编码
<code>Encoding.UTF8</code>	获取 UTF-8 格式的编码
<code>Encoding.ASCII</code>	获取 7 位 ASCII 字符集的编码
<code>Encoding.Unicode</code>	获取特定顺序的 Unicode 格式的编码

在字符串转换到字节数组的过程中，`Encoding.Default` 会将每个单字节的英文或者数字字符转换成一个字节，而把每个双字节的汉字字符转换成两个字节。而 `Encoding.Unicode` 则会将所有需要转换的部分都转换成两个字节。读者可以通过下例简单地了解一下转换的方法，以及使用 `Encoding.Default` 和 `Encoding.Unicode` 的区别，示例代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;

```

```

using System.Linq;
using System.Text;
namespace _3._4._8
{
    //定义 Program 类
    class Program
    {
        //私有成员变量
        static string s = "C#语言";
        static byte[] b1 = System.Text.Encoding.Default.GetBytes(s);
        static byte[] b2 = System.Text.Encoding.Unicode.GetBytes(s);
        static string t1 = "", t2 = "";
        //定义程序的 Main() 方法
        static void Main(string[] args)
        {
            //Main() 方法的处理过程
            foreach (byte b in b1)
            {
                t1 += b.ToString("") + " ";
            }
            foreach (byte b in b2)
            {
                t2 += b.ToString("") + " ";
            }
            //打印输出结果
            Console.WriteLine("b1.Length = " + b1.Length);
            Console.WriteLine(t1);
            Console.WriteLine("b2.Length = " + b2.Length);
            Console.WriteLine(t2);
            Console.Read();
        }
    }
}

```

运行结果如下：

```

b1.Length = 6
67 35 211 239 209 212
b2.Length = 8
67 0 35 0 237 139 0 138

```

使用 Encoding 类的 string GetString(byte[]) 或 string GetString(byte[], int, int) 方法都可以将字节数组转换成字符串，读者的选择要按照编码的实际需求决定。以下述代码为例，在 TestStringBytes() 函数中添加如下语句：

```

byte[] bs = {97, 98, 99, 100, 101, 102};
string ss = System.Text.Encoding.ASCII.GetString(bs);
Console.WriteLine("The string is: " + ss + "\n");

```

运行结果为：

```
The string is: abcdef
```

3.4.9 数值类型和字节数组之间的转换

在前面介绍的转换方法中，我们知道了各种数值型需要使用多少字节的空间来保存数据。当各种数值类型和字节数组之间转换时，编译器将某种数值类型的数据转换成字节数

组，得到的一定是相应大小的字节数组。同理，当把字节数组转换成数值类型时，这个字节数组的大小必须大于相应数值类型的字节数。一般会利用 `System.BitConverter` 类来实现各种数值类型与字节数组之间的转换。

3.4.10 不同类型之间的强制转换

在 C# 语言中，可以利用 `Convert` 进行强制转换。`Convert` 可以将一个基本数据类型转换为另一个基本数据类型。表 3.8 列出了 `Convert` 类公开的成员。

表 3.8 `Convert` 类公开成员

公共字段	<code>DBNull</code>	一个常数，表示没有数据的数据库列（即数据库为空）
公共方法	<code>ChangeType</code>	返回具有指定类型，而且其值等效于指定对象的 <code>Object</code>
	<code>Equals</code>	确定两个 <code>Object</code> 实例是否相等（从 <code>Object</code> 继承）
	<code>FromBase64CharArray</code>	将 <code>Unicode</code> 字符数组的子集（它将二进制数据编码为 base 64 数字）转换成等效的 8 位无符号整数数组。参数指定输入数组的子集以及要转换的元素数
	<code>FromBase64String</code>	将指定的 <code>String</code> （它将二进制数据编码为 base 64 数字）转换成等效的 8 位无符号整数数组
	<code>GetHashCode</code>	用作特定类型的哈希函数。 <code>GetHashCode</code> 适合在哈希算法和数据结构（如哈希表）中使用（从 <code>Object</code> 继承）
	<code>GetType</code>	获取当前实例的 <code>Type</code> （从 <code>Object</code> 继承）
	<code>GetTypeCode</code>	返回指定对象的 <code>TypeCode</code>
	<code>IsDBNull</code>	返回有关指定对象是否为 <code>DBNull</code> 类型的指示
	<code>ReferenceEquals</code>	确定指定的 <code>Object</code> 实例是否相同的实例（从 <code>Object</code> 继承）
	<code>ToBase64CharArry</code>	将 8 位无符号整数数组的子集，转换为用 Base 64 数字编码的 <code>Unicode</code> 字符数组的等价子集
	<code>ToBase64String</code>	将 8 位无符号整数数组的值转换为它的等效 <code>String</code> 表示形式（使用 base 64 数字编码）
	<code>ToBoolean</code>	将指定的值转换为等效的布尔值
	<code>ToByte</code>	将指定的值转换为 8 位无符号整数
	<code>ToChar</code>	将指定的值转换为 <code>Unicode</code> 字符
	<code>ToDateTime</code>	将指定的值转换为 <code>DateTime</code>
	<code>ToDecimal</code>	将指定值转换为 <code>Decimal</code> 数字
	<code>ToDouble</code>	将指定的值转换为双精度浮点数字
	<code>ToInt16</code>	将指定的值转换为 16 位有符号整数
	<code>ToInt32</code>	将指定的值转换为 32 位有符号整数
	<code>ToInt64</code>	将指定的值转换为 64 位有符号整数
	<code>ToSByte</code>	将指定的值转换为 8 位有符号整数
	<code>ToSingle</code>	将指定的值转换为单精度浮点数字
	<code>ToString</code>	将指定值转换为其等效的 <code>String</code> 表示形式
	<code>ToUInt16</code>	将指定的值转换为 16 位无符号整数
	<code>ToUInt32</code>	将指定的值转换为 32 位无符号整数
	<code>ToUInt64</code>	将指定的值转换为 64 位无符号整数

计算机内部的任何数据都是以二进制保存的，所以，程序员只需要关心进制转换的字符串中的结果即可。

显式转换可以替代隐式转换，但隐式转换不能替代显式转换。换句话说，可以用显式转换的地方，隐式转换也可以使用，只要按照程序员的需求自主选择就好。但需要显式转换的地方，就一定不能用隐式转换。

3.5 C#的用户自定义数据类型

C#程序使用类型声明创建新类型。类型声明指定了新类型的名字和成员。C#可由用户自定义类型，这种类型一般被称做复杂数据类型。本节将和读者一起来研究复杂数据类型。

3.5.1 认识枚举类型

枚举其实不是 C#特有的。与 C 中的枚举一样，C#中的枚举包含枚举值，以及与值关联的数字。所谓“枚举”，就是把这种类型数据可取的值一一列举出来。

枚举类型是一种特殊的值类型，是一组已命名的数值常量，常用于声明一组命名的常数。枚举的类型必须是之前已经讲到的 8 个整型类型之一。

枚举用关键字 `enum` 来声明定义。定义枚举类型的一般格式为：

```
enum 枚举名[: 数据类型]
{ 成员 1[=整型常数 1], 成员 2[=整型常数 2], ..., 成员 n[=整型常数 n] }
```

枚举类型常用于归类定义一些在编译时已知范围的，但是具体值要在执行时才能确定的常量。可以用读者习惯的描述性名称表示枚举中的整数值，比如，已知三原色是红、蓝、绿，它们同属于颜色。可以定义如下：


```
enum Color
{
    Red,
    Blue,
    Green
}
```

枚举相比普通整数常量有一定的优胜之处，这主要体现在它使代码更容易阅读理解和更安全。没有指定特定的枚举类型，那么一般会默认枚举元素的基础类型为 `int`。默认状态下，将 0 赋值给枚举对象的第一个元素，然后对每个后续的枚举元素按 1 递增。但是，也可以在初始化阶段给元素直接赋值。

具体例子如下：

```
enum Weekdays { Mon=1, Tues, Wed, Thur, Fri, Sat, Sun }
```


此时，枚举成员的值从 1 开始，而不是从 0 开始，即 `Mon` 值为 1，`Tues` 值为 2，其后依次加 1，则最后一个 `Sun` 值为 7。

 **说明：**通常允许对枚举进行整型运算。


```
enum Weekdays {Mon,Tues,Wed,Thur= -8,Fri=13,Sat,Sun}
```

此时, Mon=0, Tues=1, Wed=2, Thur=-8, Fri=13, Sat=14, Sun=15。

通常, 在需要用一组符号来表示一组整数值的情况下, 可以使用枚举。

说明: System.Enum 是引用类型, 而枚举类型又是直接继承自 System.Enum 的, 但是枚举类型却不是引用类型。其实这部分内容涉及 MISL 部分的内容, 简要来说, 枚举类型隐式的继承自 System.Enum, 程序员不能在 C# 里显式声明这种继承关系, 所以枚举类型都是值类型。


3.5.2 枚举编程示例

下面通过代码实例为读者具体介绍枚举的语法构成。代码如下:

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._5._2
{
    //定义 Program 类
    class Program
    {
        //定义 Test 枚举类型成员
        enum Test
        {
            A = 1,
            B,
            C,
            D,
            E,
            F = 10,
            G
        };
        //定义程序的 Main( ) 方法
        static void Main(string[] args)
        {
            int x = (int) (Test.A);
            int y = (int) (Test.B);
            int z = (int) (Test.F);
            int m = (int) (Test.G);
            //{0}{1}...充当占位符, 将后面的参数对号入座
            Console.WriteLine("Test.A = {0}", x);
            Console.WriteLine("Test.B = {0}", y);
            Console.WriteLine("Test.F = {0}", z);
            Console.WriteLine("Test.G = {0}", m);
            Console.Read();
        }
    }
}
```

输出结果为:


```
Test.A =1
Test.B =2
Test.F =10
Test.G =11
```

 **注意：**如果移除初始值 A=1 以及 F=10 的设定，因为枚举的首项默认的是 0，那么结果将是：

```
Test.A =0
Test.B =1
Test.F =5
Test.G =6
```

创建枚举可以节省大量的时间。使用枚举比使用无格式的整数有如下几个优势：

- ☐ 枚举可以提高程序的可维护性，有助于确保给变量指定合法的期望值；
- ☐ 枚举使代码简练、清晰；
- ☐ 枚举使代码减少录入次数，提高效率。

 **技巧：**枚举类型可以通过使用 System.Enum 的 ToString() 方法转换成字符串类型。


3.5.3 认识结构类型

结构类型也是一种值类型，它是一种堆栈（stack）分配的复杂数据类型，它不支持继承。结构主要用于创建小型的对象，并可以节省内存。

结构可以包含构造函数、常量、字段、方法、属性、索引器、运算符、事件和嵌套类型等。但是，如果同时需要上述几种成员，则应当将类型更改为类。结构的定义和类的定义基本上是一致的，语法如下：

```
//struct 为声明结构的关键字
struct StructName
{
    public int structDataMember;
    //结构中显示声明的构造函数必须带有参数
    public void StructMethod(参数列表)
    {
        //结构体实现
    }
}
```

结构名称最好是某个现有名词或名词短句。此外，也可以拥有和类一样的成员。然而，与类类型不同的是，结构是值类型，不需要在堆（heap）中分配空间。而且结构不是引用类型，所以也不支持继承。

 **说明：**结构被存在堆栈（stack）中。结构可以提高存储效能。当一个结构与类有着相同信息的条件下，结构可以大大地减少对存储空间的占用。

3.5.4 结构编程示例

在下例中，程序创建并初始化 10000 个 points。在类 Point 中需要分配 10001 个独立的


对象 (object)。代码如下:

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _3._5._4
{
    //定义 Point 类
    class Point
    {
        //公共类型成员变量
        public int x, y, z;
        //定义类的成员方法
        public Point()
        {
            x = 0;
            y = 0;
            z = 0;
        }
        public Point(int x, int y, int z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
    }
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Point[] points = new Point[10000];
            //执行对象 points 的初始化, 其中每个点的坐标值分别为 (x,y,z)=(i,i*2,i*3)
            for (int i = 0; i < 10000; i++)
            {
                points[i] = new Point(i, i * 2, i * 3);
            }
        }
    }
}
```

很明显, 这个 Point 类繁琐复杂, 当执行 10000 次初始化工作的时候会影响程序的运行速度。类的定义与初始化经常会遇到类似的困境。这时就需要结构来大显身手了。当 Point 被定义为一个结构时, 程序的执行可以简化很多。代码如下:


```
struct Point
{
    public int x, y, z;
    public Point(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

不用考虑编译器的执行效率，仅从程序的结构上就不难发现，结构的定义比类要精简颇多！结构 Point 中的坐标变量利用内联的性质实例化使程序整体得到了优化。

 **注意：**尽管可以利用结构来简化类的声明以及实现步骤，但并不是所有情况下都比类优越。作为值类型的结构在传递数据时实际上是在传递数据本身的复制值，而引用类型的类则是传递值的引用地址。所以当传递结构的时候就会比传递类要慢，数据量越大差距就越明显。

3.5.5 结构也支持方法

在一个结构的默认值的计算过程中，将所有值类型字段设置为它们的默认值，并将所有引用类型字段设置为 null，这样就产生了该结构的默认值。结构可以使用封箱和拆箱操作在结构类型和对象类型之间进行转换。而且，在结构中声明的实例构造函数必须携带参数。

 **注意：**结构的赋值是特别需要注意的。

对于结构，this 关键字有着不同的意义。结构实例构造函数和实例成员函数中的 this 也和类中的 this 含义有所不同。在类中，this 本质上是一个引用句柄值，它不可以再被赋值。而在结构中，this 更像一个结构类型的变量，它可以被赋值。

为了便于理解，下面以一个代码实例来展开论述。首先定义一个描述公司的结构 sCompany。代码如下：

```
struct sCompany
{
    public string CompanyName;
    public object CompanyMember;
    public string CompanyAddress;
    public int CompanyGradetime;
}
```

读者不难看出，这个声明同类的声明相似。在结构体当中，声明了 4 个属性变量，分别用以描述公司名称、公司成员、公司地址，以及公司的成立年份。由于使用结构的许多限制，所以读者不必通过 new 关键字对结构进行实例化。但是，如果不亲自为结构的成员 new 空间使得显式初始化，那么就永远不会完成该结构初始化。

```
sCompany s123456;
Console.WriteLine(s123456.CompanyName);
```

由于没有通过 new 关键字分配空间，上述代码无疑将会报错。为了纠正这个错误，改进程序如下列代码所示：

```
sCompany s123456 = new sCompany();
Console.WriteLine(s123456.CompanyName);
```

因为 s123456.CompanyName 是一个值类型，所以初始化默认为 0。由于结构不能创建无参数的构造函数，所以，以下代码不能编译通过：


```

struct sCompany
{
    public sCompany();
    public string CompanyName;
    public object CompanyMember;
    public string CompanyAddress;
    public int CompanyGradetime;
}


```

但是，可以利用带有参数的构造函数原型来定义，代码如下：

```

struct sCompany
{
    public sCompany(string name, object member, int number, int gradetime)
    {
        //声明 sCompany 结构中的名称、成员、地址、成立年份属性
        CompanyName = name;
        CompanyMember = member;
        CompanyAddress = address;
        CompanyGradetime = gradetime;
    }
    //类的成员变量
    public string CompanyName;
    public object CompanyMember;
    public string CompanyAddress;
    public int CompanyGradetime;
}

```

 **说明：**结构的语法中没有基类的列表。这是由于结构不能基于其他结构或类，而且它们也不能作为其他结构或类的基类。

3.5.6 结构与类有什么不同


在 C# 中，结构与类是完全不同的类型，具体区别如下：

- ☐ 结构是值类型，在堆栈（stack）上分配地址，结构之间的赋值可以创建新的结构；而类是引用类型，在堆（heap）上分配地址，类之间的赋值只是复制引用。
- ☐ 从执行效率上来讲，堆栈要比堆高得多。然而堆栈由于资源的限制，导致其不适合处理大的逻辑复杂的对象，所以结构仅适合处理小型对象，而类则可以处理某个商业逻辑。
- ☐ 结构与类都是由 System.Valuetype 派生而来的。类可派生于一个基类，也可派生于任何多的接口，结构却不可以。
- ☐ 结构不能被继承，类可以被继承。
- ☐ 结构和类都能够继承接口。
- ☐ 结构提供默认的不带参的构造函数，且不允许替换，但是允许用户自行添加带参的构造函数。相比之下，类的构造函数允许替换，带参或者不带参均可。
- ☐ 结构没有析构函数，类有析构函数。

- ❑ 结构不能使用 `virtual`、`abstract` 和 `sealed` 关键字，类可以使用。
- ❑ 结构不能有 `protected` 修饰符，类允许。
- ❑ 结构中不能初始化实例字段，类可以初始化实例字段。
- ❑ 结构可以指定字段如何在内存中布局，类不具备。

3.5.7 哪些地方应使用结构类型

通过以上的介绍，读者可以很自然地意识到结构在效率上相对于类的优越性，这主要归因于它们在底层的值类型结构。不过，由于结构对于大容量数据及高复杂度的算法处理时的局限性，导致其适用范围非常狭窄。如果用户正在从事图像色彩或是固定模式的业务处理程序的设计时，可以选用结构类型来构造这些小规模数据体。但是，由于结构实际上是一种复杂的值类型，所以在编译器中其整体上被定义为一个数据，这对所有的面向对象操作而言是不利的。

 **注意：**不要试图在结构里构造过多的方法，最好是能不定义方法，就尽量避免定义方法。


3.6 本章总结

从本章开始，读者开始真正接触 C# 语言的代码操作。在本章中，首先讲解了变量的类型以及定义，接着又为读者介绍了 C# 语言中数据类型间的转换，最后为读者详细阐述了常用的复杂数据类型，枚举和结构。

3.7 实战练习

1. 在 Visual Studio 2010 中新建一个控制台应用程序，编写代码对两个字符串进行连接操作，最后输出连接后的内容。

2. 在 Visual Studio 2010 中新建一个控制台应用程序，分别输出汉字“中”和“国”的 Unicode 码。

 **提示：**先将汉字分别保存在两个 `char` 类型的变量，然后通过类型转换的方法输出其 Unicode 码。

3. 在 Visual Studio 2010 中新建一个控制台应用程序，将字符串“Visual C#”分解到字符数组中，然后输出数组中第 8 个元素的内容。

4. 在 Visual Studio 2010 中新建一个控制台应用程序，用一个枚举类型来表示一个星期中的七天，输出枚举值的数值及其对应的字符串如下所示。

```
0 星期日
1 星期一
```


2 星期二
3 星期三
4 星期四
5 星期五
6 星期六

提示：可通过以下代码输出一个枚举量的数值及字符串值。

```
Console.WriteLine("{0:d} - {0}", week.星期日);
```

第4章 变量与表达式

在本章中，将主要介绍 C#语言中的变量与表达式的相关知识。变量或常量就是指通过指定特定位置来存放数据的存储单元。通常来说，一个变量或者常量对应着一定的数据类型，如整型和字符型等。一个以上的某种类型的变量或者常量在特定运算操作符的修饰下就形成了表达式。

4.1 常量与变量

变量和常量是所有编程语言的基础，而每一门编程语言都有自己的变量和常量的命名和使用方式。本节将对 C#语言中的变量和常量进行讲解，主要内容包括：变量和常量各自的用途、如何对变量和常量进行命名，以及如何定义和初始化变量和常量。

计算机程序处理的对象是数据，而数据是以某种特定的形式存在的。通过第3章的学习我们了解到，C#语言将数据分为不同的类型，如整型、浮点型和字符型等，它们分别表示不同范围、不同精度、不同用途的数据。

在 C#中，编译器会给每个数据指定一个存储单元，并以变量或常量的形式来命名数据的存储单元。实际上，每个变量或常量都会对应着某些特定的数据类型。本章将和读者重点讨论常量以及变量的相关知识。


4.1.1 什么是常量

常量就是指在程序编译时就已经存在并且一直保持不变的值。某种意义上讲，常量必然是已经声明的“常数”。这里的“常数”不仅指的是值类型，有时候也可以是字符串或者某些引用类型。类和结构可以将某些“常数”声明为成员。常量必须在声明时初始化。

从数据类型的角度来看，常量的类型可以是任何一种值类型或引用类型。在 C#中定义常量有两种方式，一种叫做静态常量，即编译时常量（Compile-time constant），另一种叫做动态常量，即运行时常量（Runtime constant）。静态常量用“const”来定义，动态常量用“readonly”来定义。

4.1.2 静态常量的特点

静态常量是指用 const 定义的常量，尽管静态常量不能使用 static 关键字，但可以像访问静态成员那样去访问 const 定义的常量。

 **注意：**用对象的成员方式去访问会出变异错误。静态常量相当于 C 语言里面的 #define，它必须在声明时就初始化，不能在构造函数里初始化，也不能从变量里提取值来初始化。未包含在定义静态常量的类中的表达式必须使用类名、一个句点和常量名来访问该常量。

静态常量修饰符可以是 new、public、protected、internal 和 private。

静态常量的类型 type 必须是以下之一：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、string、枚举类型或引用类型。

C# 中的常量以 const 关键字进行声明，后面接 C# 支持的数据类型之一和赋值语句。

C# 中的常量的定义语法如下：


访问修饰符 const 数据类型 常量名称 = 常量值；

示例代码如下：

```
class Months
{
    public const int months = 12;
}
```

在上例中，只要对 months 赋值，这个常量 months 的值将始终为 12，不能更改。

用 const 定义常量在类型上有很多限制。首先，const 定义常量的类型必须属于值类型，此类型的初始化不能通过 new 来完成，因此一些用 struct 定义的值类型常量不能用 const 来定义。

 **注意：**一般情况下，引用类型是不能被声明为 const 常量的，不过 string 例外。该引用类型 const 常量的值可以有两种情况，string 或 null。NET 却对 string 类型处理的方法比较特别。编译器通常采用字符串恒定性（immutable）方法处理字符串使得 string 的值具有只读特性。

4.1.3 动态常量的特点

动态常量也被称做运行时常量。相对于 const 而言，用 readonly 来定义常量要灵活很多。顾名思义，正因为系统要为 readonly 所定义的常量分配空间，即和类的其他成员一样拥有独立的空间，所以被称做动态常量。

用 readonly 关键字所定义的常量本身具有一些特点。可以在类的构造函数中设定它们。readonly 所定义的常量相当于类的成员，因此对它的声明实际上相当于为一个类新建了一个实例。正是由于这一特性使得使用 readonly 定义的常量没有 const 常量的诸多类型限制。在 C# 中，readonly 比 const 要更加灵活，可以用 readonly 关键字去定义任何类型的常量。因此，对于那些无法使用 const 来声明的常量，就可以使用 static readonly 声明。

4.1.4 该用静态常量还是动态常量

比较了静态常量和动态常量的异同之后，读者可能已经产生了这样的疑问：定义常量的时候，到底是用 const 来定义还是 readonly 呢？很多程序员为了追求性能，尽量用 const

来定义。但是，使用 `const` 会产生潜在的错误。


一般来说，假如在编译过程中，程序使用了 DLL 类库中某个类的静态常量，而程序员仅修改了类库中静态常量的值，其他接口没有发生变化，那么程序调用端是不会自动重新编译的。编译器只是直接调用新的类库。不过由于静态常量用它的值去替换常量，因此在调用端的程序也是这样进行替换。例如：

```
public const int n= 20;
```

在编译后产生的中间语言代码中所有使用该静态常量的地方 `n` 的值都改为 20 了。当类库的静态变量发生变化后，例如：

```
public const int n= 15;
```


刚才已经提到，调用端程序可以在不重新编译的情况下运行。此时程序的中间语言代码对应于静态变量 `n` 的值是 20，而新类库中的 `n` 的值是 15，因此会产生不一致。这样，程序会引发潜在的错误。

 **说明：** `readonly` 定义的常量类似于类的成员，需要根据常量地址来实现访问。因此用 `readonly` 定义常量时是不会发生如 `const` 常量般的错误，从而可以避免此类错误。

4.1.5 什么是变量

在前面的各节中，读者已经了解了常量的意义与使用方法。但是，程序员不可能只应用常量就能解决所有的问题。在实际开发过程中，经常被应用到的实际上是变量。

变量（`variable`）是指已经声明了数据类型的字符或者字符串。变量的赋值可以在定义的时候就直接完成，也可以在后续的程序执行过程中被赋值。

 **说明：** 变量中，真正存储的是数据的地址。因此在编译过程中，变量的值能够根据地址的改变而改变。

变量定义的一般格式如下：

```
数据类型 变量名;
```

在定义变量的同时对变量进行赋值，这个过程就是变量的初始化。在 C# 中，变量声明和初始化的格式如下：

```
数据类型 变量名=表达式;
```

例如：

```
int year=2008;
```

初始化一个变量的过程实际上就是对变量赋值的过程。变量赋值的实质就是将数据保存到变量中。给变量赋值可以理解为对变量初始化的简略操作，格式如下：

```
变量名 = 表达式;
```

当变量的存储地址需要改变的时候，变量的值可以通过赋值运算或者自加、自减运算来改变其值的内容。示例代码如下：



```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._1._5
{
    class Program
    {
        static void Main(string[] args)
        {
            //定义变量 a 但是不直接赋值
            int a;
            //此时编译器不允许输出, 因为变量 a 没有赋值
            //Console.WriteLine("a="+a);
            //定义变量 b 并且直接赋值
            int b = 1;
            Console.WriteLine("b=" + b);
            //通过赋值改变变量 a 的值
            a = b;
            Console.WriteLine("a=" + a);
            //通过赋值改变变量 b 的值
            b += 1;
            Console.WriteLine("b=" + b);
            //通过自加运算改变 a 的值
            a++;
            //通过自减运算改变 b 的值
            --b;
            Console.WriteLine("a=" + a);
            Console.WriteLine("a={0}, b={1}", a, b);
            Console.Read();
        }
    }
}
```

输出结果如下:

```
b=1
a=1
b=2
a=2
a=2, b=1
```

4.1.6 认识变量的实质

变量的实质是存放各种类型数据的内存单元。正是由于其存储的值可以改变, 因此编译器就规定了变量“先定义, 后使用”的准则。这样可以避免很多不必要的编译错误。定义变量的过程实际上就是为变量分配内存单元大小的过程。

说明: 通过之前的学习读者已经了解到, 不同数据类型的字节数是不同的, 因此, 定义了不同的数据类型, 也就相当于为变量分配了内存单元的最大额度。变量之后的赋值必须在这个类型的范围之内, 否则将会发生变异错误。

4.1.7 变量有哪些种类

在C#语言中把变量分为7种类型，它们分别是静态变量（static variables）、非静态变量（instance variables）、数组元素（array elements）、值参数（value parameters）、引用参数（reference parameters）、输出参数（output parameters）和局部变量（local variables）。各种变量名称以及用法总结如表4.1所示。

表 4.1 C#的变量

类 型	名 称	变 量
值类型	各种值类型	各个值类型的范围内允许值
引用类型	类类型	初始化为 null，允许对该类类型实例的引用或由该类类型派生类的实例的引用
	对象类型	初始化为 null，允许任何引用类型的对象引用或任何值类型封箱值的引用
	接口类型	初始化为 null，允许实现该接口的类类型实例的引用或实现该接口的值类型封箱值的引用
	数组类型	初始化为 null，允许该数组类型实例的引用或兼容数组类型实例的引用
	委托类型	初始化为 null，允许该委托类型实例的引用

在下面的示例代码中，给出了这几种类型变量的声明方法。

```
class A
{
    //静态变量 x
    public static int x;
    //非静态变量 y
    int y;
    void F(int[] v,int a,ref int b,out int c)
    {
        //局部变量 i
        int i=1;
        c=a+b++;
    }
}
```

在上面的变量声明中，x 是静态变量，y 是非静态变量，v[0]是数组元素，a 是值参数，b 是引用参数，c 是输出参数，i 是局部变量。

1. 静态变量

静态变量是指带有 static 修饰符声明的变量。静态变量不能在方法中定义，只能在类中定义。静态变量建议在定义时赋值。静态变量的初始值就是该变量类型的默认值。静态变量所属的类被加载调用以后，静态变量就会抑制存在，直到包含该类的程序运行结束时为止。如：


```
static int a=0;
```


2. 非静态变量

实例变量即不带有"static"修饰符声明的变量。如：

```
int a;
```

一个类的非静态变量应该在初始化时就赋值。非静态变量属于实例，而每个实例都有自己的变量，只能通过实例名来访问。由于实例变量一开始是不存在的，要通过 new 关键字分配内存空间后才能访问；而静态变量属于类的实例，不用实例化系统一开始就为它分配了内存，因此不用 new 关键字就可以直接访问。


 **注意：**在同一类中，所有的实例共享同一个静态变量，可以通过类名和实例名对静态变量访问。

3. 数组元素

数组元素是应用范围非常广的一种变量类型。关于这类变量的具体介绍，将在后面为读者详细介绍。

4. 局部变量

局部变量是指在一个独立的程序块或者独立的作用域中声明的变量，它只在声明范围内有效。这种变量随着程序执行区域的变化而生效或者失效。

 **说明：**与其他几种变量类型不同的是，局部变量不会被自动初始化，所以也就没有默认值。在进行赋值检查的时候，编译器会自动认为局部变量是没有被赋值的。

在局部变量的有效范围内，在变量定义之前就使用是不合法的，比如：

```
for(int i=0;i<=100;i++)
{
    //非法，因为局部变量 a 还没有定义
    int b=a;
}
```

经过修改以后正确的代码如下：

```
for(int i=0;i<=100;i++)
{
    //正确
    int a;
    int b=a;
}
```


5. 其他

有关值参数、引用参数和输出参数的内容都属于函数的参数，这里不做具体讨论。关于这部分的知识，将在函数的相关章节再为读者详细讲解。

4.1.8 怎样给变量命名

下面和读者讨论一下各种变量的命名规范。在介绍各种变量的命名方法之前，先为读者简要介绍一下 C#语言中常用的命名方法。

- ❑ 骆驼命名法：顾名思义是指混合使用大小写字母来构成变量和函数的名字。如 `myData`，它第 1 个单词的第 1 个字母小写，后面的单词首字母大写，形似骆驼。
- ❑ 帕斯卡（`pascal`）命名法：与骆驼命名法类似，只不过骆驼命名法是首字母小写，而帕斯卡命名法是首字母大写。例如 `MyData`。
- ❑ 匈牙利命名法：匈牙利命名法通过在变量名前面加上相应的小写字母的符号标识作为前缀，标识出变量的作用域和类型等。这些符号可以多个同时使用，顺序是先 `m_`（成员变量），然后是指针，接着是简单数据类型，最后是其他类型。匈牙利命名法的关键是：标识符的名字以一个或者多个小写字母开头作为前缀，前缀之后的是首字母大写的的一个单词或多个单词组合，该单词要指明变量的用途。

 **注意：**Windows 编程中用到的变量（还包括宏）的命名规则都采用了匈牙利命名法。譬如 `iMyData`，小写的 `i` 说明了它的形态，后面的和帕斯卡命名相同，指示了该变量的用途。

类变量也称为成员变量。通常来说，它是以骆驼命名法声明，但应以一个下划线开头。不应使用匈牙利命名法命名变量，并且应该避免使用单个字符作为变量名。

在讲述完各种命名法以后，现在开始讲解各种变量的命名规则。

在 C#中，变量的命名规则主要有如下几条：

- ❑ 变量名必须以字母或下划线开头。
- ❑ 变量名只能由字母、数字、下划线组成。
- ❑ 变量名不能包含空格、标点符号、运算符等其他符号。
- ❑ 变量名区分大小写。
- ❑ 变量名不能与 C#中的关键字名称相同。但是，可以通过在关键字前加前缀 `@` 的方法，使它变为合法的变量名。

4.2 连接的桥梁——运算符与表达式

运算符是指那些可以用来控制完成操作数完成各种类型运算的操作符号，如加号（`+`）、减号（`-`）等。用运算符作为前缀或者后缀修饰一个操作数，或者把两个以上的操作数通过运算符连成一个整体之后，就形成了表达式，例如：

```
4 * (6 + 4)
```

4.2.1 C#的运算符分类

在 C#中，运算符是指能接受一个或多个称为操作数的表达式作为输入并返回值的术语

或符号。运算符按操作数的数目来分，有一元运算符、二元运算符和三元运算符。

- 一元运算符：也叫单目运算符，主要用于只有单个操作数的表达式运算，通常只接受一个操作数的运算符。可以出现在操作数的前面，如`-n`，`--n`或者增量运算符（`++`）。
- 二元运算符：也叫双目运算符，接受两个操作数的运算符。主要作用于两个操作数的运算并始终出现在两个操作数的中间，如`x+y`。
- 三元运算符：也叫三目运算符，用于三个操作数的表达式运算。C#中只有一个三元运算符，即条件运算符“`? :` ”。

按照功能的不同，运算符又可以分为算数运算符、关系运算符、赋值运算符、逻辑运算符和位运算符。在本章中将逐一介绍这些运算符的定义以及使用方法。

表达式可以包含文本值、运算符及其操作数、方法调用，也可以仅指一个简单的名称。表达式可以使用运算符，也可以完成对方法函数的调用。应该说表达式本身所能实现的意义是非常丰富多彩的。正是由于这种灵活的机制，才使表达式既可以非常简单，也可以非常复杂。

4.2.2 算术运算符

C#语言中提供了大量运算符，这些运算符是指定在表达式中执行哪些操作的符号。很多运算符可被用户重载，因此在用户自定义的类型中可以更改这些运算符的含义。表 4.2 详细列出了所有算数运算符的定义以及作用。

表 4.2 算术运算符

算术运算符	作 用
<code>+</code>	用于执行加法运算，如果操作的是字符串，则作为字符串的连接运算符
<code>-</code>	减法运算
<code>*</code>	乘法运算
<code>/</code>	除法运算，并得到商
<code>%</code>	除法运算后的取余
<code>++</code>	操作数加 1
<code>--</code>	操作数减 1
<code>~</code>	将一个数按位取反

算术运算符包括基本算术运算符`+`、`-`、`*`、`/`、`%`和自增、自减运算符`++`、`--`，其中自增、自减运算符`++`、`--`将在 4.2.10 节中单独介绍。

算术运算符的操作对象通常是值类型的变量或常数。在下面的示例中，将具体讲解这些算术运算符的用法，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._2._2
{
```

```
//定义 Program 类
class Program
{
    static void Main(string[] args)
    {
        int a=8, b=4, c,d;
        //此时, a=11
        a=a+b;
        //此时, b=2
        b=b-1;
        //此时, c=22
        c=a*b;
        //此时, d=5
        d=a/b;
        //此时, x=2
        int x=c%d;
        //打印输出结果
        Console.WriteLine ("a={0}", a );
        Console.WriteLine ("b={0}", b );
        Console.WriteLine ("c={0}", c );
        Console.WriteLine ("d={0}", d );
        Console.WriteLine ("x={0}", x );
        Console.Read();
    }
}
```

输出结果为:

```
11
2
22
5
2
```

- 算术加法、算术减法、算术乘法: C#中的算术加法、算术减法与算术乘法适用于所有值类型之间的运算。与 C/C++语言一样, 同类型之间的这 3 种算术运算按照一定的运算顺序依次完成。例如:


```
1.1+2-4*4=-8.9
```

- 算术除法: 在 C#中, 算术运算的除法与 C 语言中一样会舍掉小数, 例如:

```
5/2=2
```

- 算术取余: 取余是用除数除以被除数, 取最大整数商, 然后求余数, 余数可以是小数。例如:

```
5%2=1
```

 **注意:** 算术运算符 (+、-、*、/) 必须要考虑到参与运算的各个操作数的值域问题。算术运算结果有可能会超出操作数类型的可能范围。

一般情况下, 算数运算异常情况如表 4.3 所示。

表 4.3 算术运算溢出

算 术 运 算	导 致 异 常
整数算术溢出	OverflowException
整数被零除	DivideByZeroException
浮点算术溢出	不引发异常
浮点被零除	不引发异常
小数算术溢出	OverflowException
小数被零除	DivideByZeroException

除算术运算符以外，整型之间的强制转换也会导致溢出。例如，将 long 强制转换为 int，就会导致溢出错误。整型之间强制转换溢出时，也受 checked 或 unchecked 执行的限制。

在运算过程中可以进行类型转换，转换的形式为：

(目标类型) 数值或变量

例如：

```
char A = (char) 65;
```

表示把数值 65 转换成字符'A'。

4.2.3 关系运算符

关系运算符包括“==”、“!=”、“<”、“>”、“<= ”、“>= ”和 is，主要用于比较两个操作数的大小。比较的结果是一个布尔值，即要么为 true（真），要么为 false（假）。

is 用来判断一个变量是否是某一类型，例如，

```
//判断变量 s 是否为字符串类型
s is string
```

表 4.4 主要列出了关系运算符的作用。

表 4.4 关系运算符

关系运算符	作 用
>	检查一个数是否大于另一个数
<	检查一个数是否小于另一个数
>=	检查一个数是否大于等于另一个数
<=	检查一个数是否小于等于另一个数
==	检查两个数是否相等
!=	检查两个数是否不等
is	判断一个变量是否是某一类型

在下面的示例中，将具体讲解这些关系运算符的用法。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
```

```

using System.Text;
namespace _4._2._3
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write ("请输入一个数字: ");
            string str=Console.ReadLine();
            double x=double.Parse(str);
            // ">" 关系运算
            if (x > 10)
            {
                Console.WriteLine("您输入的数字大于 10");
            }
            else
            {
                Console.WriteLine("您输入的数字小于或等于 10");
            }
            Console.Read();
        }
    }
}

```

程序的运行结果如下:


```

请输入一个数字:
15
您输入的数字大于 10

```

4.2.4 一般赋值运算符

赋值运算符为“=”，是一个二元运算符，其作用就是将一个数据赋给一个变量。赋值中，右操作数表达式所属的类型必须可隐式地转换为左操作数所属的类型。运算将右操作数的值赋予左操作数指定的变量、属性或索引器元素。赋值表达式的结果是赋予左操作数的值，结果的类型与左操作数相同，且始终为值类别。

 **注意：**如果左操作数为属性或索引器访问，则该属性或索引器必须具有 set 访问器，否则，将发生编译时错误。

赋值表达式的格式:

变量=表达式;

在 C# 中，可以对变量进行连续赋值，这时，运算顺序是自右向左。例如:

```

int a, b, c;
//此时, a,b,c 值都变为 15
a=b=c=15;

```

对引用类型的数组元素的赋值需要运行时检查，以确保所赋的值与数组实例兼容。在

下面的示例中，因为 ArrayList 的实例不能存储在 string[] 的元素中，程序最后的赋值导致引发 System.ArrayTypeMismatchException 异常，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._2._4
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            string[] str = new string[10];
            object[] obj = str;
            //赋值正确
            obj[0] = null;
            //赋值正确
            obj[1] = "Hello";
            //将会引发异常 ArrayTypeMismatchException
            obj[2] = new ArrayList();
            Console.Read();
        }
    }
}
```

4.2.5 复合赋值运算符

在赋值运算符当中，有一类 C/C++/C# 独有的复合赋值运算符。它们实际上是一种缩写形式，变量与表达式先进行运算符所要求的运算，再把运算结果赋值给参与运算的变量。凡是二目运算都可以用复合赋值运算符来简化表达。采用复合赋值运算符会降低程序的可读性，但这样却可以使程序代码简单化，并能提高编译的效率。表 4.5 详细列出了所有复合赋值运算符及其功能。


 **说明：**通过复合赋值运算符可以更为简便地改变表达式以及变量的值。

表 4.5 复合赋值运算符

符 号	功 能	符 号	功 能
+=	加法赋值	<<=	左移赋值
-=	减法赋值	>>=	右移赋值
*=	乘法赋值	&=	位逻辑与赋值
/=	除法赋值	=	位逻辑或赋值
%=	模运算赋值	^=	位逻辑异或赋值

复合赋值运算的一般形式为：

变量 复合赋值运算符 表达式

$a=a+42$ 可写为 $a+=42$;

$y=y/(x+9)$ 可写为 $y/=x+9$ 。

在下面的示例中, 将具体讲解这些复合赋值运算符的用法, 代码如下:

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._2._5
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义变量并进行初始化
            int a = 124;
            float f = 1.24f;
            char ch = 'f';
            uint i = 11;
            Console.WriteLine("int a={0}, \nfloat f={1}, \nchar ch={2}, \nuint i={3}", a, f, ch, i);
            //进行不同的运算, 并打印输出结果
            a += 5;
            Console.WriteLine("\na+=5 运算后 a 的值是: " + a);
            a -= 90;
            Console.WriteLine("a -= 90 运算后 a 的值是: " + a);
            a *= 4;
            Console.WriteLine("a *= 4 运算后 a 的值是: " + a);
            a /= 4;
            Console.WriteLine("a /= 4 运算后 a 的值是: " + a);
            a %= 6;
            Console.WriteLine("a %= 6 运算后 a 的值是: " + a);
            f += 1;
            Console.WriteLine("f +=1 运算后 f 的值是: " + f);
            //注意, int 与 float 型的变量不能进行位移操作
            ch >>= 1;
            Console.WriteLine("ch >>= 1 运算后 ch 的值是: " + ch);
            i <<= 1;
            Console.WriteLine("i<<= 1 运算后 i 的值是: " + i);
            Console.ReadLine();
        }
    }
}
```

输出结果为:


```
int a=124,
float f=1.24,
char ch=f,
uint i=11

a+=5 运算后 a 的值是: 128
a -= 90 运算后 a 的值是: 48
a *= 4 运算后 a 的值是: 152
a /= 4 运算后 a 的值是: 50
```




```
a %= 6 运算后 a 的值是: 2
f +=1 运算后 f 的值是: 2.24
ch >>= 1 运算后 ch 的值是: 4
i<<= 1 运算后 i 的值是: 22
```

通过上面的复合赋值运算符，读者可能会问，到底 $a=a+5$ 与 $a+=5$ 有没有区别？答案是有的。对于 $a=a+5$ ，表达式 a 被计算了两次；而对于复合运算符 $a+=5$ ，表达式 a 仅计算了一次。一般来说，这种区别对于程序的运行没有太大影响，但是当表达式作为函数的返回值时，函数就被调用了两次，而且如果使用普通的赋值运算符，也会加大程序的运行，使效率降低。关于函数部分的知识将在以后的具体章节再做说明。

说明：与 $+$ 运算符一样， $+=$ 运算符也可以用于字符串操作。

4.2.6 逻辑运算符

逻辑运算符包括 $\&$ 、 \wedge 、 $|$ 、 $\&\&$ （与）、 \parallel （或）、 $!$ （非）。 $\&\&$ 、 \parallel 为二元运算符，实现条件与、条件或运算。 $!$ 为一元运算符，实现逻辑非运算。

说明： $\&\&$ 是比较运算符，一般用于条件判断；而 $\&$ 是位运算符，直接用于位运算。

逻辑运算符的操作对象可以是逻辑量（true 或 false）或关系表达式。逻辑运算的结果也是一个布尔值，即要么为 true（真），要么为 false（假）。具体的应用将在介绍位运算符中详细讲解。表 4.6 详细列出了所有逻辑运算符的名称、作用以及结合方向。

表 4.6 逻辑运算符

逻辑运算符名称	运算符表达式	结 合 方 向
逻辑与	$\&$	左
逻辑异或	\wedge	左
逻辑或	$ $	左
逻辑非	$!$	左
条件与	$\&\&$	左
条件或	\parallel	左

左结合意味着运算符是从左到右进行运算的，如算术运算符；右结合意味着所有的运算是从右到左进行的，如赋值运算符。右结合的表达式必须等其右边的运算计算出来之后，才把结果放到左边的变量中。

逻辑运算符的应用方法总结如下。

1. !运算符

这个运算符能够实现对表达式的条件判断，若表达式值为 true，则运算结果为 false；若表达式值为 false，则运算结果为 true。

例如：

$!(1<2)$ 的运算结果为 false。

!(2<1)的运算结果为 true。

在下面的示例中，将具体讲解 “!” 运算符的用法，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._2._6
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine ("请输入一个数字: ");
            string str=Console.ReadLine();
            double x = double.Parse(str);
            //逻辑非运算 “!”, x<=0 不满足时
            if (! (x<=0) )
            {
                Console.WriteLine("您输入的数字是正数");
            }
            else if ((x==0) )
            {
                Console.WriteLine("您输入的数字是 0");
            }
            else
            {
                Console.WriteLine("您输入的数字是负数");
            }
            Console.Read();
        }
    }
}
```

这段程序的输出结果是：

```
请输入一个数字:
0
您输入的数字是 0
```

2. 表达式1 && 表达式2

这个表达式实现的操作是：只有当表达式 1 和表达式 2 的值均为 true 时，运算结果才为 true；否则，结果为 false。

例如：

(4<4)&&("abc"!="xyz")的运算结果为 false。

(4>4)&&("abc"!="xyz")的运算结果为 true。

在下面的示例中，将具体讲解 “&&” 运算符的用法，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
```



```

using System.Text;
namespace _4._2._6
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine ("请输入一个数字: ");
            string str=Console.ReadLine();
            double x=double.Parse(str);
            //逻辑与运算 "&&"，当 x>=1 和 x<10 同时满足时
            if( (x>=0) && (x<100) )
                Console.WriteLine("您输入的数字大于等于 0 且小于 100");
            else
                Console.WriteLine("您输入的数字小于 0 或者大于等于 10");
            Console.Read();
        }
    }
}

```

这段程序的输出结果是：

```

请输入一个数字:
24.444
您输入的数字大于等于 0 且小于 100

```

3. 表达式1 || 表达式2

表达式 1 和表达式 2 的值其中有一个为 true，运算结果就为 true；只有当表达式 1 和表达式 2 的值均为 false 时，运算结果才为 false。

例如：

('y!='n')||(4.28>5.0)的运算结果为 true。

('y=='n')||(4.28>5.0)的运算结果为 false。

结合以上的讲解，下面给读者列出一个综合性的示例代码，代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._2._6
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("请输入一个数字: ");
            string str = Console.ReadLine();
            double x = double.Parse(str);
            //逻辑或运算 "||"
            if (x > 0 || x < 0)
            {
                Console.WriteLine("您输入的数字不是 0");
                Console.Read();
            }
        }
    }
}

```

```

        return;
    }
    if ((x == 0))
    {
        Console.WriteLine("您输入的数字是 0");
        Console.Read();
        return;
    }
}
}

```

这段程序的输出结果是：

```

请输入一个数字：
0
您输入的数字是 0

```

4.2.7 复习二进制知识

任何数据信息在计算机中都是以二进制形式保存的。位运算是指对数据按二进制位进行运算。位运算符和移位运算符永远不会导致溢出。

在介绍位运算符之前，需要简要介绍一下计算机基础知识中，关于原码、反码以及补码的相关背景知识。


在日常生活中，读者熟悉并且经常使用的是十进制数。可是在计算机中数值的表示形式为机器数，计算机只能识别 0 和 1，即二进制数。

根据前面的介绍可以知道，C#语言的值类型包括多种类型，只有有符号的整数才有原码、反码和补码。所有的整数分为无符号数与有符号数两种。0 在计算机中储存为 00000000，对于正数，我们依然可以像无符号数那样换算，00000001~01111111 依次表示 1~127。那么这些数对应的二进制码就是这些数的原码。负数的原码和它的绝对值所对应的原码相同，简单地说就是，绝对值相同的数原码相同。反码就是把负数的原码各个位按位取反，是 1 就换成 0，是 0 就换成 1。

计算机储存有符号的整数时，是用该整数的补码进行储存的。0 的原码、补码都是 0，正数的原码、补码可以特殊理解为相同，负数的补码是它的反码加 1。补码就是在反码的基础上加 1，即 -1 的补码是 11111110+1=11111111。在补码表示中，最高位为符号位。正数的符号位为 0，负数为 1。补码的规定如下：

对正数来说，最高位为 0，其余各位代表数值本身（以二进制表示），如 +42 的补码为 00101010。

对负数而言，把该数绝对值的补码按位取反，然后对整个数加 1，即得该数补码。如 -42 的补码为 11010110（00101010 按位取反 11010101+1=11010110）。

 说明：C#中可以用 111111 表示 -1 的补码，这也是补码与原码和反码的区别所在。用补码表示数，0 的补码是唯一的，都为 00000000。而在原码、反码表示中，+0 和 -0 的表示是不唯一的，关于这部分内容就不详细讨论了。感兴趣的读者可自行参见相应的书籍。

4.2.8 二进制的位运算符

在讨论完相关的背景知识后,开始正式介绍位运算符。位运算符包括&(与)、|(或)、^(异或)、~(取反)、<<(左移位)、>>(右移位)。位运算符中,除~以外,其余均为二元运算符。

位运算符的操作数可以是整型变量或整型常数。可以对整数、布尔和枚举类型进行这3种操作。

1. 按位“与”运算

按位“与”运算的规则如下:

- ☐ $0 \& 0 = 0$;
- ☐ $0 \& 1 = 0$;
- ☐ $1 \& 0 = 0$;
- ☐ $1 \& 1 = 1$ 。

不难看出,只有当两个二进制位均为1时,计算结果才为1;否则,结果均为0。

例如: $2 \& 3$

2 的二进制表示	0000 0010
& 3 的二进制表示	0000 0011

0000 0010

计算结果为2。

2. 按位“或”运算

按位“或”运算的规则如下:

- ☐ $0 \mid 0 = 0$;
- ☐ $0 \mid 1 = 1$;
- ☐ $1 \mid 0 = 1$;
- ☐ $1 \mid 1 = 1$ 。

只有当两个二进制位均为0时,计算结果才为0;否则,结果均为1。

例如: $2 \mid 3$,

2 的二进制表示	0000 0010
3 的二进制表示	0000 0011

0000 0011

计算结果为3。

3. 按位“异或”运算

按位“异或”运算的规则如下:

- ☐ $0 \wedge 0 = 0$;

- $0^1=1$;
- $1^0=1$;
- $1^1=0$ 。

当两个二进制位相异时，计算结果为 1；当两个二进制位相同时，计算结果为 0。

例如： 2^3

2 的二进制表示	0000 0010
^ 3 的二进制表示	0000 0011

0000 0001

计算结果为 1。

4. 按位取“反”运算

规则如下：

- $\sim 0=1$;
- $\sim 1=0$ 。


可以看出，当二进制位为 0 时，计算结果为 1；当二进制位为 1 时，计算结果为 0。

例如： ~ 6

6 的二进制表示	$\sim 0000 0110$
----------	------------------

1111 1001

计算结果为-7。

说明：按位“取反”是一元运算。

5. <<（左移位）运算与>>（右移位）运算

位移运算就是指对二进制整数的操作位的移动，由于此运算的结果而被清空的位上则填补 0。在位数没越界的条件下，将一个无符号数值向左移一位等同于将它乘以 2，右移一位相当于除以 2。

下面的示例中，将整数 1 向左移 10 位。

$x=1<<10$

此运算的结果为 $x=1024$ 。这是因为十进制的 1 等于二进制的 1，二进制的 1 向左移 10 位是二进制的 1000000000，而二进制的 1000000000 就是十进制的 1024。

通过以上介绍了解到，位移计算相当于对一个整数进行 2 的 N 次幂运算。如下所示，可以按照位移的特性来快速实现程序的某些特定运算。

左移位：

$<<1=*2$;

$<<2=*4$;

$<<4=*8$;

$<<4=*16$;

.....

右移位：


```
>>1=/2;
>>2=/4;
>>4=/8;
>>4=/16;
.....
```

用<<对整数做2的乘除法非常快，有很多编译器可在内部对其进行优化。在下面的示例中，将具体讲解这些位运算符的用法。示例程序如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._2._8
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("请输入两个整数进行逻辑运算");
            //输入两个整数
            int A = int.Parse(Console.ReadLine());
            int B = int.Parse(Console.ReadLine());
            //条件与
            int andR = A & B;
            Console.WriteLine("{0} & {1} = {2}", A, B, andR);
            //条件或
            int orR = A | B;
            Console.WriteLine("{0} | {1} = {2}", A, B, orR);
            //条件异或
            int noR = A ^ B;
            Console.WriteLine("{0} ^ {1} = {2}", A, B, noR);
            //取反
            Console.WriteLine("~ {0:x8} = {1:x8}", A, ~A);
            Console.ReadLine();
        }
    }
}
```

输出结果为：

```
请输入两个整数进行逻辑运算
1
2

1&2=0
1|2=4
1^2=4
~00000001=fffffffe
```

4.2.9 有逻辑判断功能的三元运算符

条件运算符（ ? : ）是唯一的一个三元运算符，它需要3个参数。它可以计算一个

条件，如果条件为真，就返回一个值；如果条件为假，则返回另一个值。其一般格式为：

```
关系表达式? 表达式 1: 表达式 2;
```

如果“关系表达式”的值为 true，则条件运算表达式的值为“表达式 1”的值；否则，为“表达式 2”的值。例如：

```
int min, a=5, b=9;
min=(a<b) ? a : b ;
```

?: 运算符是 if...else 语句的快捷方式。它通常被用于较大表达式的一部分，在此处使用 if...else 语句是不协调的。例如：

```
string now = new Date();
Date greeting = "Good" + ((now.getHours() > 17)?" evening.":" day.");
```

在该例子中，如果是下午 6 点以后，则创建一个包含“Good evening.”的字符串。使用 if...else 语句的等价代码如下：

```
var now = new Date();
var greeting = "Good";
if (now.getHours() > 17)
    greeting += " evening.";
else
    greeting += " day.";
```

4.2.10 自增和自减运算符

自增(++)、自减(--)运算符都是一元运算符，它们的主要功能是：

- (1) 自增(++)运算符将操作数的值自动加 1；
- (2) 自减(--)运算符将操作数的值自动减 1。

这两个运算符的功能相近，都是将数值变量的值加 1 或减 1，用户只能将这类操作符应用于变量而不能应用于常量。二者的区别是前缀式先将操作数增 1（或减 1），然后取操作数的新值参与表达式的运算。后缀是先将操作数增 1（或减 1）之前的值参与表达式的运算，到表达式的值被引用之后再做加 1（或减 1）运算。

下面的代码给出了这种运算符的实际应用，示例如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _4._2._10
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            int x = 4, y;
```



```

        //实现不同的自加和自减过程,并输出结果
        y = x++;
        Console.WriteLine("x={0},y={1}", x, y);
        y = ++x;
        Console.WriteLine("x={0},y={1}", x, y);
        y = x--;
        Console.WriteLine("x={0},y={1}", x, y);
        y = --x;
        Console.WriteLine("x={0},y={1}", x, y);
        Console.Read();
    }
}

```


程序运行结果如下:

```

x= 5,y= 4
x= 6,y= 6
x= 5,y= 6
x= 4,y= 4

```

可以看出,自加和自减运算符可在变量名前,也可在变量名后,即都可以用于前缀和后缀的形式,但含义并不相同。对于前缀的形式,变量先作自加或自减运算,然后将运算结果用于表达式中;而对于后缀的形式,变量的值先在表达式中参与运算,然后再做自加或自减运算。

 **说明:** 通常,程序员更喜欢使用整型变量的自加或自减运算。

4.2.11 应该先进行什么运算

当一个表达式含有多个运算符(操作符)时,运算符的优先级决定运算的执行顺序。对于表达式 $x+y*z$,编译器将先算 $y*z$,再与 x 相加。因为“*”运算符比“+”运算符的优先级高。当一个操作数出现在两个具有相同优先级的运算符之间时,运算符的结合性决定运算的执行顺序。

表达式中的运算符按照称为运算符优先级的特定顺序计算。在C#中,一共有48个常用的运算符,根据它们所执行运算的特点和它们的优先级,可将它们归为7个等级。

- ☐ 单元运算符和括号;
- ☐ 常规算术运算符;
- ☐ 位移运算符;
- ☐ 关系运算符;
- ☐ 逻辑运算符;
- ☐ 赋值运算符;
- ☐ 后缀单元运算符。

这7个等级的优先级别依次递减。具体情况如表4.7所示。

表 4.7 运算符优先顺序

优 先 级	运 算 符	结 合 顺 序
高 ↑ 低	++、--（前置）、()、+、-（作为单元运算符时）、!、~	从右到左
	*, /, %	从左到右
	+, -	从左到右
	<<, >>	从左到右
	<, >, <=, >=, is, as	从左到右
	==, !=	从左到右
	&, ^,	从左到右
	&&,	从左到右
	?:	从右到左
	=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=	从右到左
	++、--（后置）	从左到右

当表达式中出现两个具有相同优先级的运算符时，它们根据结合性进行计算。左结合运算符按从左到右的顺序计算。例如， $x * y / z$ 计算为 $(x * y) / z$ 。右结合运算符按从右到左的顺序计算。赋值运算符和三元运算符 (?:) 是右结合运算符。其他所有二元运算符都是左结合运算符。然而，C# 标准没有指定何时执行表达式中的增量指令的“设置”部分。

例如，下面的代码示例输出为 6：

```
int num1 = 5;
num1++;
System.Console.WriteLine(num1);
```

而下面的代码示例输出却是未定义的：

```
int num2 = 5;
//没有定义
num2 = num2++;
System.Console.WriteLine(num2);
```

因此，建议不要使用后一个示例。可以在表达式两侧使用括号来强制在计算其他表达式之前计算该表达式。

4.3 关 键 字


在第3章的学习过程中，读者应给已经了解了关键字的基本构成。在本章学习了变量以及表达式的相关知识之后，相信读者已经掌握了C#中各种变量与表达式的定义及使用方法。在这里必须再介绍一下C#的关键字构成。因为在之后的学习过程中，读者应该可以逐步自行设计编写程序了，那么请记住一点：当给任何的类、变量或者表达式命名的时候，请远离这些关键字。关键字是不能作为任何类名或变量名使用的。

4.4 本章总结

在本章中，首先介绍了常量和变量的定义和用途，以及常量和变量的常用类型，并介绍了在 C#语言中的常用命名方法。在第二节中，介绍了在 C#语言中的常用运算符，以及这些运算符的结合顺序和优先级。最后介绍了将变量与运算符相结合所得到的表达式。

4.5 实战练习

1. 在 Visual Studio 2010 中创建一个控制台应用程序，在 Main 函数中编写代码，让用户输入两个整型值，然后输出这两个数相加、相乘、相减的结果。
2. 在 Visual Studio 2010 中创建一个控制台应用程序，在 Main 函数中编写代码，让用户输入两个整型值，然后判断这两个数的大小，并输出比较的结果。
3. 在 Visual Studio 2010 中创建一个控制台应用程序，在 Main 函数中编写代码，让用户输入一个数，然后判断这个数是小于 0、大于 0 还是等于 0，并输出判断的结果。
4. 在 Visual Studio 2010 中创建一个控制台应用程序，在 Main 函数中编写代码，让用户输入一个表示员工年龄的整数，程序判断用户输入的年龄值是否正确。

提示：员工年龄应该在 16~60 岁之间。


第5章 程序控制语言

从本章开始，将学习有关语句（Statements）的相关知识。C#直接借鉴了 C 和 C++中绝大多数的语句，因此 C#中的很多语句其实和 C++的相似。只是 C#中的语句对一些错误进行了修正，并且增加了一些新的语句，例如经常用于处理多线程的 foreach 语句，以及用于控制数学计算及算术转换溢出检查操作及会话的 checked/unchecked 语句。

5.1 C#有哪些种类的语句

根据功能不同，C#语句可以分为以下几种：

- ❑ 选择语句：提供了在两条或多条路径间选择执行的途径。
- ❑ 循环语句：使程序在所给条件满足的情况下反复执行某一段代码。
- ❑ 跳转语句：使函数内的程序无条件地改变控制权，在程序间进行控制转移，主要用于进行无条件跳转。
- ❑ 注释语句：不参与程序运行，但是会对程序中的类、方法、变量的用途做出解释，使程序代码方便易读。
- ❑ 标签语句：在一行语句里被标示以一个前缀，主要用于支持 goto 跳转。
- ❑ 声明语句：用来对变量初始化赋值。
- ❑ 表达式语句：计算表达式的值。
- ❑ 块语句：界定局部变量的作用范围。
- ❑ 空语句：没有执行任何操作。
- ❑ using 语句：用于获取已有的资源。
- ❑ try...catch 语句：用于定义及捕获异常。
- ❑ checked/unchecked 语句：用于控制数学运算及类型转换，向指定目标类型操作时溢出状况检查。
- ❑ lock/unlock 语句。lock 语句可以为调用的对象设置排他锁，当程序运行以后，则可以根据 unlock 语句释放 lock 内容。

说明：在 C#中，控制语句使用的范围最为广泛。它的主要作用是控制程序中各语句的执行顺序，主要分为 3 种，即选择语句、循环语句和跳转语句。

5.2 选择语句让程序具有智能

5.2.1 选择语句的作用

在 C#语言中，有两种选择结构语句：if...else 语句和 switch 语句。它们的作用就是根据所给条件（关系、布尔表达式）是否满足，决定是否执行后面的语句。

选择语句提供了在两条或多条路径间选择执行的途径。选择语句有两种主要的类别，即两分支选择和多分支选择。当设定条件成立时执行某些语句，条件不成立时则跳过这些语句或执行其他语句。选择语句主要应用于选择结构的判断及控制，有两路分支和多路分支两种情况，对应的选择语句有两路出口的 if 语句和多路出口的 Select Case 语句。本节将主要介绍这两种选择语句。

5.2.2 认识 if 语句

在 C#语言中，if 语句是最常用的选择语句。if 语句的实质是选择执行语句体中布尔表达式的结果值。如果 if 语句的布尔值判断是肯定的，那么执行 if 语句后面的语句或者语句块；如果一条 if 语句的布尔值判断是否定的，那么它可能包括 else 语句块。示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._2._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            {
                //进行条件判断
                if (args.Length == 0)
                {
                    Console.WriteLine("没有携带命令行参数");
                }
                else
                {
                    Console.WriteLine("携带了命令行参数");
                }
            }
            Console.Read();
        }
    }
}
```


示例代码的输出结果是：

没有携带命令行参数

if 语句主要展示了程序中输出两种不同的信息结果。这种判断结果基于是否满足 if 的条件判断。当 if 语句的条件判断为真时，它之后的语句体中所有语句都成为判断结果执行的语句。在本段示例代码中，程序在执行 if() 语句的条件判断时满足了条件表达式 `args.Length == 0` 的条件，所以程序直接执行 if 之后的语句，输出执行结果并且跳出了程序。而 else 则未被执行，所以 else 之后的显示语句也没有显示。

常见的 if 语句有 4 种形式：

- ☐ 单条选择 if 语句；
- ☐ if...else...语句；
- ☐ 复杂 if...else...语句；
- ☐ 嵌套 if...else...语句。

 说明：if 中的判断语句必须是布尔型常量或者变量。用户不能像在 C/C++ 中一样使用一个数值表达式。

5.2.3 单分支 if 语句

单条选择 if 语句为程序增加了一个判断条件，使程序在符合条件时将执行 if 语句中的代码。单条选择 if 语句的具体形式如下所示。

```
if(条件表达式)
{
    语句块;
}
```

下面的示例程序使用单条选择 if 语句编写，读者可结合示例对单条选择 if 语句的使用方法进行理解，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._2._3
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义变量
            int i;
            Console.WriteLine("请输入一个整数: ");
            string str = Console.ReadLine();
            //获得输入的值
            i = int.Parse(str);
            //对 i 进行判断
            if (i > 100)
            {
```



```

        Console.WriteLine("您输入整数的值大于 100");
    }
    Console.Read();
}
}
}

```


示例代码执行结果如下：

```

请输入一个整数：
235
您输入的整数的值大于 100

```

以上程序仅进行了 if 条件的判断，这会使程序在执行阶段带来一个问题，假如要输入的整数是小于 100 的，那么程序将不会做出任何逻辑处理。当程序员需要 if 判断之外的其他控制的时候，就需用到 if 语句的其他几种形式了。

 **注意：**在当前这一段代码中，if 语句判断之后就跳出了程序，程序对不满足 if 语句条件表达式的其余情况不做任何处理。

5.2.4 二分支 if 语句

if…else…语句为条件判断后的 boolean 函数的 false 值增加了一条路径。使程序能够对不满足 if 语句条件表达式的其余情况做出处理。具体格式如下：

```

if(条件表达式) {语句块 1}
else {语句块 2}

```

下面的示例程序使用 if…else…语句编写，读者可结合示例对 if…else…语句的使用方法进行理解，代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._2._4
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义变量
            int a, b, max;
            Console.WriteLine("请输入一个整数：");
            string str1 = Console.ReadLine();
            //将字符串 str1 转换成 double 类型以进行比较
            a = int.Parse(str1);
            Console.WriteLine("请输入第二个整数：");
            string str2 = Console.ReadLine();
            b = int.Parse(str2);
            //对 a 和 b 的值进行判断，执行不同的处理

```

```

        if (a > b)
        {
            max = a;
        }
        else
        {
            max = b;
        }
        Console.WriteLine("两个整数中最大值为: {0}", max);
        Console.Read();
    }
}


```

程序的输出结果为:

```

请输入一个整数:
23
请输入第二个整数:
55
两个整数中最大值为: 55

```

说明: 通过这段代码可以看出, 利用 if...else...语句能很好地解决之前仅有 if 判断时所遇到的问题。

实际项目中的判断不会像示例代码这样简单明了, 如果需要处理复杂的逻辑判断, 就需要使用其他方法了。


5.2.5 多分支 if 语句

当程序需要进行复杂的逻辑判断时, 前面所介绍的两种 if 语句就远不能够满足要求, 这个时候读者可以选择使用本节中介绍的复杂 if...else...语句, 或者使用在 5.2.6 节中介绍到的嵌套 if...else...语句。具体格式如下:

```

if (条件表达式 1) { clause 语句块 1 }
else if (条件表达式 2) { clause 语句块 2 }
else if (条件表达式 3) { clause 语句块 3 }
//.....
else if (条件表达式 n-1) { clause 语句块 n-1 }
else { clause 语句块 n }

```

注意: 本节中介绍的复杂 if...else...语句和将在 5.2.6 节中介绍到的嵌套 if...else...语句, 在通常情况下是可以互换使用的。

下面的示例程序将使用复杂 if...else...语句进行编写, 读者可结合示例对复杂 if...else...语句的使用方法进行理解。代码如下:

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


```



```

namespace _5._2._5
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义变量
            int a, b, c, max;
            //获得 a、b、c 三个整数
            Console.WriteLine("请输入一个整数: ");
            string str1 = Console.ReadLine();
            a = int.Parse(str1);
            Console.WriteLine("请输入第二个整数: ");
            string str2 = Console.ReadLine();
            b = int.Parse(str2);
            Console.WriteLine("请输入第三个整数: ");
            string str3 = Console.ReadLine();
            c = int.Parse(str3);
            //对 a、b、c 三个整数的数值进行判断
            if ((a > b) && (a > c))
            {
                max = a;
            }
            else if ((b > a) && (b > c))
            {
                max = b;
            }
            else
            {
                max = c;
            }
            Console.WriteLine("三个整数的最大值为: {0}", max);
            Console.Read();
        }
    }
}

```

 说明：通过 else if 的衔接，if…else…语句的处理模块可以实现多选项的复杂判断。

5.2.6 if 语句多层嵌套

除了前面所介绍的各种方式外，if…else…语句还可以进行多层嵌套，即在大括号内添加另一个 if…else…判断。那样就能够实现更多的选择判断。

下面将为读者展示一段略为复杂一些的 if 嵌套语句。这段代码能够判断前台输入的数据中是否含有数字 0。为了实现一个对 0 的判断逻辑，程序运用多重条件表达式进行判断。示例代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._2._6

```

```
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义变量
            int a, b, c, max;
            //获得 a、b、c 三个整数
            Console.WriteLine("请输入一个整数: ");
            string str1 = Console.ReadLine();
            a = int.Parse(str1);
            Console.WriteLine("请输入第二个整数: ");
            string str2 = Console.ReadLine();
            b = int.Parse(str2);
            Console.WriteLine("请输入第三个整数: ");
            string str3 = Console.ReadLine();
            c = int.Parse(str3);
            //对 a、b、c 三个整数的值进行判断
            if ((a > b) && (a > c) && (a != 0))
            {
                max = a;
                int mul = b * c;
                //实现 if 语句嵌套
                if ((b != 0) && (c != 0))
                {
                    Console.WriteLine("三个整数都是正整数");
                    Console.WriteLine("最大值是: " + max);
                    Console.WriteLine("其余两个数的乘积是: " + mul);
                }
            }
            else
            {
                Console.WriteLine("最大值是: " + max);
                Console.WriteLine("其余两个数的乘积是: " + mul);
            }
        }
        else if ((b > a) && (b > c) && (b != 0))
        {
            max = b;
            int mul = a * c;
            //if 语句嵌套
            if ((a != 0) && (c != 0))
            {
                Console.WriteLine("三个整数都是正整数");
                Console.WriteLine("最大值是: " + max);
                Console.WriteLine("其余两个数的乘积是: " + mul);
            }
            else
            {
                Console.WriteLine("最大值是: " + max);
                Console.WriteLine("其余两个数的乘积是: " + mul);
            }
        }
        else if (c != 0)
        {
            max = c;
            int mul = a * b;
            //if 语句嵌套
```



```

        if ((a == 0) || (b == 0))
        {
            Console.WriteLine("三个整数都是正整数");
            Console.WriteLine("最大值是: " + max);
            Console.WriteLine("其余两个数的乘积是: " + mul);
        }
        else
        {
            Console.WriteLine("三个整数都是正整数");
            Console.WriteLine("最大值是: " + max);
            Console.WriteLine("其余两个数的乘积是: " + mul);
        }
    }
    else
    {
        Console.WriteLine("三个整数都是 0");
    }
    Console.Read();
}
}
}

```

这段代码实现得比较繁琐，在学习完所有的控制语句之后，读者可以思考简化数据结构的复杂度。这从另一个侧面展现了嵌套 if…else…语句的特性。在实际操作中，运用嵌套 if 语句的最大好处是提高了程序的可读性，但是程序的简洁性也会受到直接的影响。

代码根据录入值输出结果之一如下：

```

请输入第一个整数：
0
请输入第二个整数：
0
请输入第三个整数：
0
三个整数都是 0

```

5.2.7 switch 多分支选择语句

switch 语句即多分支选择语句，它是另一种选择语句，根据表达式的值来决定执行哪一段代码。switch 语句可以得出一组语句，它们与一组已经定义好的表达式相匹配，如果没有匹配的出口时那么将执行 default 语句。

说明：if 语句多用于判断某个变量或表达式的值在某一范围内时执行某段代码，而 switch 语句多用于判断某个变量或表达式的值等于某个值时执行某段代码。

switch 语句的具体定义格式如下：

```

switch(表达式)
{
    case 常量表达式 1 :
        语句块 1
        break;
    case 常量表达式 2 :


```

```

        语句块 2
        break;
    //.....
    case 常量表达式 n :
        语句块 n
        break;
    default :
        语句块 n+1
        break;
}

```

switch 语句首先计算控制表达式的值，假如 case 标签后的常量表达式符合控制语句所求出的值，内含语句被执行，否则，在没有常量表达式符合控制语句时，就执行在 default 标签内的内含语句。如果没有一个符合 case 标签，且没有 default 标签，则控制转向 switch 语段的结束端。

说明：case 后边的常数不一定要从 1 或者 0 开始。实际上，只要是不重复的正整数都可以成为 case 标签的常量表达式。

switch 语句基于一个数值或者字符选择控制流程，它允许在任意数目的语句或语句组中选择其一。同 C/C++ 语言一样，当 case 语句不能立即逐条执行的时候，switch 语句不允许控制流程跳出语句体。也就是说，用户必须使用 break 语句和 switch 语句配套。

5.2.8 switch 语句编程示例

下面的示例程序将使用 switch 语句进行编写，读者可结合示例对 if...else...语句的使用方法进行理解，代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._2._8
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //输出提示信息
            Console.WriteLine("1. 苹果");
            Console.WriteLine("2. 香蕉");
            Console.WriteLine("3. 鸭梨");
            Console.WriteLine("5. 橘子");
            Console.Write("\n 请在这几种水果当中选择一个选项，输入所选数字:");
            //获得用户输入
            string str = Console.ReadLine();
            int i = int.Parse(str);
            //根据用户选择进行不同的处理
            switch (i)
            {

```



```

        case 1:
            Console.WriteLine("\n 您选择的是:苹果");
            break;
        case 2:
            Console.WriteLine("\n 您选择的是:香蕉");
            break;
        case 3:
            Console.WriteLine("\n 您选择的是:鸭梨");
            break;
        case 5:
            Console.WriteLine("\n 您选择的是:橘子");
            break;
        default:
            Console.WriteLine("\n 您选择的选项不存在");
            break;
    }
    Console.Read();
}
}
}

```

代码根据录入值输出结果之一如下:

程序输出结果为:


```

1. 苹果
2. 香蕉
3. 鸭梨
5. 橘子

```

请在三种水果当中选择一项: 1

您选择的是: 苹果

 **技巧:** 程序员不一定非要按照固有的模式, 执行从 case 标签到 default 的选择顺序。在适当的时机可以使用 goto 语句跳到程序的另外一个部分。但是必须强调一点, 作为初学者, 还是尽量避免过多使用 goto 跳转语句。因为这种语句往往会产生一些意想不到的后续问题, 甚至会影响整个程序的逻辑判断路径。

5.3 用循环语句进行重复劳动

在 C# 语言中, 有 4 种循环结构语句, 它们分别是 do 语句、while 语句、for 语句, 以及 foreach 语句。它们的作用就是在所给条件满足的情况下反复执行某一段代码。

5.3.1 do 循环语句

do 循环实质上应该被称为 do-while 循环。一个 do 循环可以一次或者多次有条件地执行一条语句或者一个语句块。

do-while 循环的使用格式如下:

```
do
{
    语句块
}
while(条件表达式)
```

下面的示例程序将使用 do 循环语句进行编写，读者可结合示例对 do 循环语句的使用方法进行理解，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._3._1
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            string str;
            //定义 do 循环处理
            do
            {
                str = Console.ReadLine();
            }
            while (str != "Out");
        }
    }
}
```

编译程序并运行，程序会一直执行 Console.ReadLine() 函数，一直到用户输入 Out 并且按 Enter 键才退出 do 循环结构。

5.3.2 while 循环语句

当 while 循环的条件判断语句为真时，它可以有条件地、零次或多次执行一条语句或者一个语句块。while 循环的使用格式如下：

```
while(条件表达式)
{
    clause 语句块
}
```

下面的示例程序将使用复杂 while 循环语句进行编写，读者可结合示例对 while 循环语句的使用方法进行理解。在这一程序中，将利用 while 循环找出一个数组的初始值，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._3._2
```




```

{
    //定义 Program 类
    class Program
    {
        //定义 Arr() 方法
        static int Arr(int value, int[] Arr)
        {
            int i = 0;
            //定义 while 循环, 输出 Arr 数组中的数据
            while (Arr[i] != value)
            {
                if (++i > Arr.Length)
                    throw new ArgumentException();
            }
            return i;
        }
        static void Main(string[] args)
        {
            Console.WriteLine(Arr(5, new int[] { 0, 1, 2, 3, 5, 5 }));
            Console.Read();
        }
    }
}

```

输出结果为:

5

 **技巧:** 在控制台输出语句的后面添加 “Console.Read();” 可使程序暂停执行, 帮助读者观察运行结果。

5.3.3 for 循环语句

for 循环用来循环执行一系列的语句, 直到函数最初用于检验的布尔表达式得出错误的判断值为止。for 循环是一种精简以后的 if...else 模式的循环判断语句。for 循环一般适合应用于事先知道循环次数的情况。for 循环的使用格式如下:

```

for (表达式 1; 表达式 2; 表达式 3)
{
    语句块
}

```

实质上表达式 1; 表达式 2; 表达式 3 分别对应 initializer、condition、iterator 这 3 项。initializer 既可以是一个或多个循环控制变量, 也可以是初始化循环控制变量的表达式。condition 代表有一个或多个循环控制条件语句。iterator 是改变循环控制变量的域值。

for 语句是 C# 中使用频率最高的循环语句。在下面的示例代码中将使用 for 循环语句求 1~100 的累加和, 如下所示。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

```


```

namespace _5._3._3
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //i 为循环变量, sum 为求和变量
            int i;
            int sum = 0;
            //使用 for 循环进行累加
            for (i = 1; i <= 100; i++)
            {
                sum += i;
            }
            //打印输出结果
            Console.WriteLine("1+2+...+100={0}", sum);
            Console.Read();
        }
    }
}

```

输出结果为:

```
1+2+...+100=5050
```

 注意: for 循环函数的 3 个参数都是可选的, 理论上并不一定完全具备。但是假如默认控制条件, 程序就可能产生一个死循环, 要用跳转语句(break 或 goto)才能退出。

for 语句执行次序如下:

- (1) 按书写顺序执行循环控制变量, 可以同时为循环控制变量赋初值。
- (2) 测试循环控制条件语句中的条件是否满足。
- (3) 若没有循环控制条件语句项或没有条件满足, 则执行一遍内嵌语句, 按预定好的规律改变循环控制变量的值, 然后继续执行第 2 步, 直至循环条件全部满足。
- (4) 若条件不满足, 则 for 循环终止, 程序跳出 for 循环函数体。

for 语句也可以嵌套, 即在一个 for 循环体内可以包含另一个 for 循环, 这样可以帮助程序员完成大量重复性、规律性的工作。在下面的代码示例中, 通过嵌套 for 语句求“1!+2!+...+10!”的和, 如下所示。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._3._3
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            int i, j, temp = 1, sum = 0;
            for (i = 1; i <= 10; i++)
            {

```




```

        //嵌套的 for 循环
        for (j = 1; j <= i; j++) temp = temp * j;
        sum = sum + temp;
        temp = 1;
    }
    //打印输出结果
    Console.WriteLine("1!+2!+...+10!={0}", sum);
    Console.Read();
}
}
}

```

输出结果为:

```
1! +2! +...+10! =5037913
```

 **技巧:** 嵌套 for 语句的运行将消耗很大的资源, 所以当可以使用非嵌套 for 语句时, 就尽量使用非嵌套 for 语句。

5.3.4 foreach 循环语句

foreach 循环语句是 C# 引入的一种新的循环类型。使用 foreach 遍历的时候其实是转换为一个一次的 while 循环判断, 只不过这个遍历的对象必须是可枚举的类型。其具体格式为:

```

foreach (类型 标识 in 表达式)
{
    嵌入语句
}

```

在下面的示例代码中将使用 foreach 循环语句判断一个整数数组中奇数和偶数的个数。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._3._4
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            int a = 0, b = 0;
            int[] arr = new int[] { 0, 1, 2, 5, 7, 8, 11 };
            //使用 foreach 循环获取 arr 数组中的数据
            foreach (int i in arr)
            {
                //对数组中的数据进行处理
                if (i % 2 == 0)
                {
                    b++;
                }
            }
        }
    }
}

```

```

        else
        {
            a++;
        }
    }
    Console.WriteLine("a=" + a);
    Console.WriteLine("b=" + b);
    Console.Read();
}
}

```

编译程序并运行，输出结果如下所示：

```

a=5
b=3

```

5.4 用跳转语句改变程序流程

跳转语句的作用是使函数内的程序无条件地改变控制权，在程序间进行控制转移，主要用于进行无条件跳转。跳转语句主要包括 `break` 语句、`continue` 语句、`return` 语句、`goto` 语句、`throw` 语句等 5 种。其中，前 4 种与 C++ 里的语义相同，`throw` 语句本节不做详尽介绍，将在介绍异常处理的相关章节为读者详细阐述。

5.4.1 用 `break` 语句跳出循环

`break` 语句常用于跳出 `switch` 选择结构或 `while`、`do...while`、`for`、`foreach` 循环结构。它主要用于终止最内层 `while`、`do`、`for` 和 `switch` 语句的执行。当程序遇到这一语句之后，执行紧接在被终止执行结构语句后面的语句。

`break` 语句的使用格式：

```
break;
```

在下面的示例代码中将使用 `while` 循环语句和 `break` 语句判断任给的一个整数 `n`，是否为素数。若 `n` 不能被 2、3……`n-1` 中任意一个数整除，则 `n` 为素数。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._4._1
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //打印提示信息
            Console.WriteLine("判断一个数是否为素数");
            Console.Write("请输入一个正整数: ");

```



```

//获得输入的整数
int n = int.Parse(Console.ReadLine());
int i = 1;
//使用 while 循环进行处理
while (++i < n)
{
    if (n % i == 0)
    {
        Console.WriteLine("{0}不是素数", n);
        //使用 break 跳出循环
        break;
    }
}
if (i == n)
{
    Console.WriteLine("{0}是素数", n);
}
Console.Read();
}
}

```

在上面的程序中，通过 `break` 语句，可以终止 `if` 循环继续执行，返回循环的布尔判断表达式部分重新开始循环。

输出结果如下：


```

判断一个数是否为素数
请输入一个正整数：
23
23 是素数

```

5.4.2 用 `continue` 语句进入下次循环

`continue` 语句通常用于跳出 `while`、`do...while`、`for` 或 `foreach` 循环中的本次循环，即跳过循环体中的剩余语句而强制执行下一次循环。

说明：`continue` 语句仅能用来终止当前进行的这次循环的最内层的循环体。在 `while` 和 `do` 循环结构中，它将控制权转至对真值条件的计算。它与前面提到的 `break` 语句不同，它并不终止整个循环的执行，而仅仅终止当前这一次循环的运行。

`continue` 语句的使用格式：

```
continue;
```

在下面的示例代码中将使用 `if` 选择语句和 `continue` 语句判断 100 以内能被 7 整除的整数 `n`，示例代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._4._2
{

```

```
//定义 Program 类
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("100 以内能被整除的整数如下: ");
        //使用 for 循环进行处理
        for (int i = 1; i <= 100; i++)
        {
            if (i % 7 != 0)
            {
                //使用 continue 结束本次循环处理
                continue;
            }
            Console.Write("{0}, ", i);
        }
        Console.Read();
    }
}
```

if 语句负责判断控制程序是否满足判断条件, 如果满足 *i* 可以被 7 整除, 那么可以继续执行 for 循环语句体内的其余语句, 即可以输出 *i* 的数值; 如果 *i* 不能被 7 整除, 那么 continue 语句则强制停止 for 循环后面的语句, 而开始重新执行 for 循环的下一次循环。

输出结果如下:

```
100 以内能被整除的整数如下:
7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98
```

5.4.3 用 return 语句返回

return 语句为包含 return 语句的成员函数返回控制权。一个没有表达式的 return 语句仅能被用于没有返回值的函数成员, 而拥有表达式的 return 语句仅能被应用在有返回表达式的函数成员。

return 语句的使用格式:

```
continue;
```

在下面的示例代码中将计算半径为 5 的圆的面积, 并通过 return 语句获取计算结果。示例代码如下:

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._4._3
{
    //定义 Program 类
    class Program
    {
        static double Area(int r)
        {
            //计算圆的面积 area
```



```


        double area;
        area = r * r * Math.PI;
        //返回 area 的值
        return area;
    }
    static void Main(string[] args)
    {
        //半径=5
        int r = 5;
        Console.WriteLine("半径为{0}的圆的面积是{1:0.00}", r, Area(r));
        Console.Read();
    }
}

```

在静态函数 `Area()` 中，已经预先执行了圆的面积的计算公式。这个计算结果被保存在变量 `area` 中。当程序调用主函数 `Main()` 的时候，就可以通过调用 `Area()` 来实现圆的面积的计算。

代码运行结果：

```
半径为 5 的圆的面积是： 78.55
```

 说明：在本段程序中，还运用了函数之间传递参数等知识。这些内容将在以后的章节中逐一讲解，读者在此可不必关注。

5.4.4 用 goto 语句跳到指定行

在 `goto` 语句中必须使用到标签。标签是由一个标识符和紧跟其后的冒号组成的语句。标签语句提供了一种被标记的前缀，`goto` 语句可以被用于把控制语句转到标签语句中。为了跳至一个特定的标签，`goto` 语句必须在该标签的范围内。`goto` 语句直接跳转到标签所标识的语句中执行语句块中的内容。

`goto` 语句的常用格式为：

```
goto 标识符
```

在下面的示例代码中将通过使用 `goto` 语句完成简易的“Hello world”程序。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._4._4
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {

            goto H;
            W: Console.WriteLine("world");
        }
    }
}


```

```

        Console.Read();
        return;
H: Console.WriteLine("Hello, ");
    //使用 goto 跳转到标记语句处
    goto W;
}
}
}

```

在上面的代码中，第一个标签语句把控制语句转换到标签 H，第一部分的信息已经被写好了，这时下一条语句将控制程序转换到了 W。最后方法被返回。

 **注意：**C#允许函数内小规模跳转，但是不支持跨函数的跳转。因此要求标号与 goto 语句处在同一个函数中。

5.4.5 用 goto 语句跳出 switch 语句

goto 语句也能用来跳出 switch 语句。switch 语句中包含两个对 switch 语句有效的跳转语句。

☐ goto case: 跳转到所说明的标签。

☐ goto default: 跳转到 default。

goto 语句被用于 switch 语句跳转的格式为：

```

goto case 常量表达式
goto default

```

下面的示例演示一个简单的价格计算函数，演示了如何将 goto 语句与 switch 语句结合使用，代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._4._5
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //打印提示信息
            Console.WriteLine("箱子尺寸:\n1=小号\n2=中号\n3=大号");
            Console.Write("请输入您的选择: ");
            //获取用户输入
            string str = Console.ReadLine();
            int n = int.Parse(str);
            int cost = 0;
            //根据用户选择进行处理
            switch (n)
            {
                case 1:
                    cost += 35;

```



```

        break;
    case 2:
        cost += 35;

        goto case 1;
    case 3:
        cost += 50;
        //跳转到指定的语句处
        goto case 1;
    default:
        Console.WriteLine("没有这个选项");
        break;
    }
    //对 cost 的值进行判断
    if (cost != 0)
    {
        Console.WriteLine("该规格箱子售价为{0}", cost);
        Console.WriteLine("您的操作已经完成, 谢谢!");
    }
    Console.Read();
}
}
}

```

示例输出如下:

```

箱子尺寸:
1=小号
2=中号
3=大号
请输入您的选择:

```

输入:


```
2
```

输出:

```

该规格箱子售价为 70
您的操作已经完成, 谢谢!

```

 **说明:** 与 C++ 不一样, 在 Switch 语句中, C# 不允许从一个开关部分继续到下一个开关部分 (即所谓的直达功能)。程序员可任意排列所有标签, 实际上这并不会影响程序的运行结果。即使把 default 标签放在其他所有标签的前面, 也需要利用必要的控制语句才能确定程序的执行顺序, 这是由于 C# 取消使用直达所造成的。

5.4.6 用 goto 语句跳出一层嵌套循环

goto 语句还可以用来跳出一层嵌套循环。在下面的示例中, 将查找用户输入的数据是否存在于一个整数数组中, 如果查找到, 则使用 goto 语句跳出循环。具体代码如下所示。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;

```

```
using System.Linq;
using System.Text;
namespace _5._4._6
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            {
                //定义变量并进行初始化
                int x = 200;
                int count = 0;
                string[] array = new string[x];
                //数组初始化:
                for (int i = 0; i < x; i++)
                {
                    array[i] = (++count).ToString();
                }
                Console.Write("请输入一个数字, 用以查找数组: ");
                //输入一个数字
                string str = Console.ReadLine();
                //开始查找
                for (int i = 0; i < x; i++)
                {
                    if (array[i].Equals(str))
                    {
                        //跳转到指定语句处
                        goto Found;
                    }
                }
                Console.WriteLine("{0} 没有被找到.", str);
                //进行跳转
                goto Finish;
                //跳转标记
            Found:
                Console.WriteLine("{0} 已经被找到.", str);
                //跳转标记
            Finish:
                Console.WriteLine("查找过程结束");
            }
            Console.Read();
        }
    }
}
```

示例输出如下:

请输入一个数字, 用以查找数组:

输入:

2

输出:

2 已经被找到
查找过程结束

⚠注意: goto 语句可以跳出一个语句块, 不过不可用来跳进一个语句块、跳出一个类, 或退出 try...catch 语句中的 finally 块。

5.4.7 用 throw 语句抛出异常

throw 语句是一种抛出语句。通常, 读者见到 throw 语句一般是在 try...catch...throw...finally 的程序块中, 它作为其中的一个组成部分。try 语句负责捕获程序块运行过程中所产生的异常。throw 语句主要的作用是引发从 System.Exception 类派生一个对象。throw 语句的基本形式为:

throw 表达式;

下面列出一个示例程序, 代码如下:


```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._4._7
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
    //定义 Stack 类
    public class Stack
    {
        //私有成员变量
        private Node first = null;
        //定义 Empty 属性
        public bool Empty
        {
            get
            {
                return (first == null);
            }
        }
        //定义 Pop() 方法
        public object Pop()
        {
            //对 first 的值进行判断
            if (first == null)
            {
                //抛出异常
                throw new Exception("不能在一个空队列中执行冒泡排序.");
            }
            else
            {
                object temp = first.Value;
                first = first.Next;
            }
        }
    }
}
```

```

        return temp;
    }
}
//定义 Push() 方法
public void Push(object obj)
{
    first = new Node(obj, first);
}
//定义 Node 类
class Node
{
    public Node Next;
    public object Value;
    public Node(object value) : this(value, null) { }
    public Node(object value, Node next)
    {
        Next = next;
        Value = value;
    }
}
}

```

在这段程序中，仅实现了 `throw` 的异常抛出功能。

 说明：这段代码中实现了面向对象编程语言的动态联编等技术内容。

5.5 用注释语句让代码意图更明了


在 C# 中，注释语法主要有以下 3 种形式：

- ☐ `//` 注释语句；
- ☐ `/*` 注释语句 `*/`；
- ☐ `///` 注释语句。

5.5.1 普通注释语句

微软 C# 继承了 C++ 的注释方法，利用 “`//`” 对代码行进行注释，也可以用 “`/*` `*/`” 实现跨行注释。有了这两种注释方法，开发人员可以方便地对代码进行各种注释操作。

以 “`//`” 开始的注释语句其有效范围仅从符号处至该行末尾，而 “`/**/`” 语法适合跨行注释。

 技巧：可以在一个程序体最初的开始部分使用 “`/**/`” 对整个文件的相关资料做简单地描述。

在下面的简单示例的程序中，使用了两种简单的注释方法，示例代码如下：

```

/*
作者：DN
版本：1.1

```




```

时间: 2012/10/19
描述: 对 Program 类进行合理的扩展
*/
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _5._5._1
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            /*
            //这是一个嵌套注释, 在 C#中是不合法的
            */
            System.Console.WriteLine("Hello,World");
        }
    }
}

```

通过以上的示例代码, 相信读者已经基本了解了 C# 的注释语句的使用规则。介绍完前两种比较简单通用的注释语法之后, 接下来着重介绍 C# 专用的一种注释方法——“///”, 并且对这种注释方法的语法本质进一步讨论。

 **注意:** 在 C# 中嵌套注释被默认为不合法的。尽管有些编译器可以自动处理这些嵌套的注释语句, 但程序员应该在编程中养成一个良好的习惯, 尽量避免这种情况的发生。

5.5.2 可生成帮助文档的注释语句

C# 中的 “///” 注释方法使用原则上与原有的 “//” 相兼容, 也是一种单行注释方法。但是这种新增加的 “///” 注释方法却与 Visual Studio 2010 进行了完美的集成, 使 C# 的注释语句功能更加强大。

下面是一个使用 “///” 注释方法的例子, 具体代码如下所示。

```

/*
作者: DN
版本: 1.2
时间: 2012/10/19
描述: 进一步扩展 Test 类
*/
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;

```

```

using System.Text;
namespace _5._5._2
{
    ///<Summary>
    ///一个 Program 类
    ///</Summary>
    class Program
    {
        ///<Summary>主入口方法</Summary>
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello, World");
        }
    }
}

```

在以上代码中，注意到增加了一个<Summary>的标识符。这是由于 C#在编译过程中会生成一个中间的 XML 数据文件，这个 XML 文件中包含了所有的“///”形式的注释文字信息。如果每个类或方法名前都具有“///”形式的注释文字信息，那么这个 XML 文件就可以作为一个类的说明手册。

在 Visual Studio 2010 平台下，.NET Framework 提供的强大功能可以把这个 XML 数据文件转化成一个说明文件。当一个开发团队集体开发项目的时候，程序员只要清楚地写好“///”语法下的注释语句，系统就可以自动产生一个详细设计文档。

在 Visual Studio 2010 中选择“项目”|“属性”命令，打开如图 5.1 所示的界面，切换到“生成”选项卡，选中“XML 文档”，并在右侧的文本框中输入生成的 XML 文件名称。



图 5.1 生成

接着单击切换到“调试”选项卡，显示如图 5.2 所示的界面，在“命令行参数”右侧文本框中输入“/doc”，表示在编辑时生成 XML 文档。



图 5.2 调试

设置好以上两项后,关闭属性设置界面。接着对项目进行生成,系统将对文档中以“///”做注释的内容进行整理,最后生成一个位于 debug 文件夹中的 XML 文档,打开该文档如图 5.3 所示。

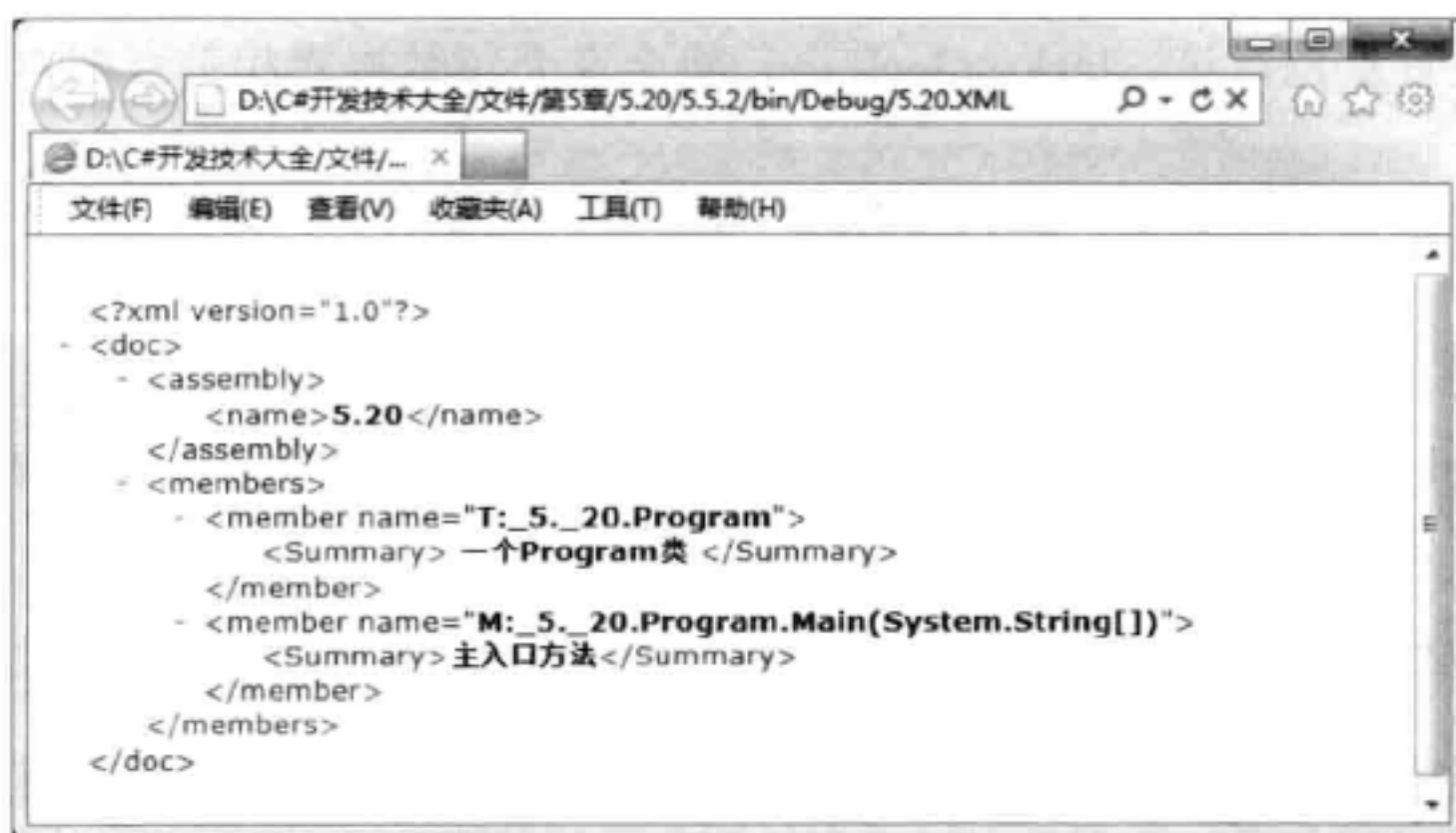


图 5.3 生成的 XML 文档

在这个 XML 文件中可以发现,所有的注释都被 Visual Studio 2010 分门别类地进行了整理,这些有条理的信息可以让同一团队中其他的人员对其一目了然。然而,这种注释的主要作用并不仅是为用户产生注释文档,更重要的是,它可以提供在编程中的智能提示作用。通过此功能,开发团队中的其他人员需要调用其他人编写的一段程序时,甚至可以不打开程序的源代码,就可以利用 Visual Studio 2010 提供的强大功能来遍历类文件中的所有属性及方法,并且根据所写的注释而得到这些属性与方法的说明。

综上所述,微软在 Visual Studio 2010 中新增加的这种注释方法,给团队开发提供了更加方便的条件,来加强团队成员之间的联系,同时个体编程人员也从中得到了更多的好处。

5.6 本章总结

C#中选择控制语句是进行程序编写的基础。本章主要介绍了如何使用 C#控制语句，其中包括选择语句的用法、循环语句的用法、跳转语句的用法，以及注释语句的用法。

由于 if 语句本身具备的简洁易懂的特性，使得它在计算机代码编写领域被极为广泛地应用。当 if 的选择分支过多而判断条件相对简单的时候，可以考虑用 switch 语句来高效地完成工作。switch 语句尤其适用于判断条件为常量表达式的情况。

循环语句中，do 循环与 while 循环具备成为死循环的特质，所以建议初学者要极其慎重地使用。相对而言，for 循环由于其自身数据结果方面的出色表现，所以受到了众多程序员的喜爱。C#中独有的 foreach 循环更能够帮助程序员结束以往令无数人头痛不已的数组遍历，以及多对象初始化等问题。当然，任何一种循环语句都有一定的语法限制，最佳的方案应该是综合的结合起来才好。

最后介绍的跳转语句需要读者在以后的实际操作中细细体会了。

只有将这些不同功能的控制语句有机地结合起来，才能真正体会到 C#的强大语言威力及其高效的灵活性。

本章开始已经为读者介绍了 C#语句的构成以及特点，所有的 C#语句都有适当的使用场合。由于 try...catch 语句涉及异常的捕获及抛出，本章暂时不作为重点内容阐述了。

check/uncheck 语句主要应用于多线程技术，这部分知识点还需要读者多多储备基础知识，所以本章没有重点讨论。lock/unlock 语句将会在介绍数据库相关章节再详细讨论。

5.7 实战练习

1. 在 Visual Studio 2010 中编写一个控制台应用程序，求 100 以内能同时被 3 与 7 整除或者被 5 除余 2 的所有数，并将这些数输出。

2. 华氏温度通过公式 $C = \frac{5}{9}(F - 32)$ 转换成摄氏温度。在 Visual Studio 2010 中编写一个控制台应用程序，从键盘上输入华氏温度，利用公式计算后输出摄氏温度，要求保留一位小数。

3. 在 Visual Studio 2010 中编写一个控制台应用程序，分别用 if 语句与 switch 语句求下列函数的值。

$$Y = \begin{cases} |x| & x < 0 \\ x & 0 \leq x < 10 \\ 3 - \frac{1}{5}x & 10 \leq x < 20 \\ 3x - 10 & 20 \leq x < 40 \\ 0 & x > 40 \end{cases}$$

4. 要将 100 元的大钞换成 1 元、5 元、10 元的小钞，若每种钞票数目大于 0，共有多少种换法？在 Visual Studio 2010 中编写一个控制台应用程序，输出各种换法（每行显示 4 组）。

第 6 章 函数与方法


通过前面几章的学习，相信读者已经掌握了 C# 的基本语法知识，并且可以利用不同的控制语句来实现程序的编写。然而，作为一名程序员，仅掌握以上的知识及技能是远远不够的。要想正确使用 C# 这门面向对象的编程语言，还必须掌握函数与方法的相关知识。本章就重点介绍它们的功能及使用方法。

6.1 函数是 C# 的基本结构

6.1.1 函数与方法

函数是所有计算机语言都会涉及的一种语法结构。一个计算机程序实质上就是通过各种函数的定义与调用来实现各种各样的功能。在面向对象的概念中，函数这种定义大部分被方法所取代，在类中声明的函数被称为方法。方法可以直接访问其所在类中的元素。在本书中，两种概念是等价的。读者可以将方法理解为用 `void` 关键字修饰的函数。在 C# 中，函数通过变量、函数参数和返回值来交换数据或者控制命令。按照函数是否携带形式参数可以分为有参数函数和无参数函数。

在 C# 中，函数的定义需要两项必备的条件，即函数的参数与返回值。一般情况下，使用一个函数时应该注意其参数列表和返回值，这两个条件是一个函数与外界交互的根本。本节将主要介绍它们的定义与使用。接下来，开始介绍函数的返回值。

 **注意：**在 C# 中，函数的定义需要两项必备的条件，即函数的参数与返回值。

6.1.2 无参和有参函数

函数可以根据是否有参数而分为有参函数和无参函数两大类，而每种类型中又可以按照是否有返回值进行分类。函数主要有以下几种表达形式。

(1) 无参函数的表达形式如下：

```
类型标志符 函数名()  
{  
    语句块  
}
```

例如：

```
void test()
```

```
{  
    int i = 0;  
}
```

“类型标志符”用来指定函数的返回值的类型，当函数没有返回值时，“类型标志符”为 void。

(2) 有参函数的表达形式如下：

```
类型标志符 函数名(参数列表)  
{  
    语句块  
}
```

例如：


```
int test(int x ,int y)  
{  
    return x+y ;  
}
```

6.2 函数的必备件：参数与返回值

在 C# 中，函数的定义需要两项必备的条件，即函数的参数与返回值。

6.2.1 参数有什么用

函数的参数是指写在函数名称后面的、圆括号内的常量值、变量、表达式或函数。函数的参数通常被称作形式参数，简称“形参”。在没有执行函数调用之前，形参并不占用任何的内存存储单元。形式参数的类型说明可在函数之后紧跟的圆括号（）之内，也可以在函数体{}内。只有当引用或调用该函数时，通过定义所传递的参数被称为实际参数，即“实参”。实参可以是变量或表达式中的任何一种，形参不能是 const 关键字修饰的常量值。形参与实参最大的联系就是实现函数中命令或者值的传递。只有实参与形参的类型相同或赋值兼容，才能实现它们之间的参数传递。

说明：通过函数的参数，程序员可以清晰、快速地了解到该函数会对什么数据做出什么处理。因此，指定了参数的函数可以提高代码的可读性。

在 C#中，函数调用传递的参数可以分为 4 类：值参数、引用参数、输出参数和数组参数。

6.2.2 值参数的使用

值参数（Value Parameter）形如 function（形式参数），是函数默认的参数类型，主要负责传递值类型变量。在 C#中，传递参数主要有两种方式，其一是传递数值，其二是传递地址。值参数就是实现值传递的媒介。传值是比较常用的一种参数传递方式。

当函数只传递一个参数的时候，示例代码如下：


```


/*只带一个参数的函数*/
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._2._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //new 一个类 Class1 的实例
            Class1 class1 = new Class1();
            //传递实际参数字符串“传递一条消息”
            class1.show("传递一条消息");
        }
    }
    //定义 Class1 类
    class Class1
    {
        //定义函数并且指定 string 型参数 message
        public void show(string message)
        {
            Console.WriteLine(message);
            Console.ReadLine();
        }
    }
}

```

上述代码的输出结果为：

传递一条消息

在 C# 中，值参数类型是经常被用到的，C# 中默认以传值的方式传递参数。读者只需要掌握实际参数的类型必须与形式参数的类型匹配或者相兼容就可以了。

 **注意：**值参数在传值过程中，即使在方法函数中被改变了其变量值，其本身还是维持初始化时的原始赋值。

6.2.3 引用参数的使用

引用参数关键字为 `ref`，形如 `function (ref 形式参数)`，主要传递实际参数的引用指针，用以辅助执行传递地址的操作。就其本质来讲，其实是在函数中直接对实参进行操作，而并非复制一个值。当函数需要返回两个或两个以上的值时，可以利用引用参数采用传递地址的方式来传递参数。使用 `ref` 类型参数的函数可以实现对外部传递过来的参数进行加工。

示例代码如下：

```

/*引用参数 ref 实现传递地址*/
//声明使用的命名空间

```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._2._3
{
    //定义 Program 类
    class Program
    {
        static public void Arr(ref int[] arr)
        {
            //设定数组初始值, arr[0]的值从变为, arr[2]的值从变为, arr[4]的值从变为
            arr[0] = 100;
            arr[2] = 6634;
            arr[4] = 0;
        }
        static void Main(string[] args)
        {
            int[] arr = new int[8] { 1, 2, 3, 4, 6, 6, 7, 8 };
            //参数 arr 是维数为{8}的数组
            //arr[0]=1, arr[1]=2,, arr[2]=3, arr[3]=4, arr[4]=6, arr[6]=6,
            //arr[6]=7, arr[7]=8
            Program.Arr(ref arr);
            //输出 arr 数组中的数据
            for (int i = 0; i < arr.Length; i++)
            {
                Console.WriteLine(arr[i]);
                Console.ReadLine();
            }
        }
    }
}

```


这段代码的输出结果为:

```

100
2
6634
4
0
6
7
8

```

在这段代码中,所有的函数方法其实都在一个类中。程序首先从入口主程序进入后,先定义并初始化了一个6维的整型数组 arr[],其中 arr[0]=1, arr[1]=2, arr[2]=3, arr[3]=4, arr[4]=6。然后程序调用已经定义好的带参函数 Arr(ref int[] arr),在函数内部,由于传递了数组的值,所以数组的维度依然是6。但是,由于在函数 Arr(ref int[] arr)内, arr[0]这时被赋值为100,因此数组的各个维度值变为 arr[0]=100, arr[1]=2, arr[2]=6634, arr[3]=4, arr[4]=0, arr[5]=6, arr[6]=7, arr[7]=8。通过 for 循环逐一输出了数组的值。

说明:读者可以在 static public void Arr(ref int[] arr){} 函数内部依次改变数组 arr[] 各个维度的数值,试验一下 ref 参数传递数组的效果。

学习了值参数与引用参数以后,读者已经初步掌握了值传递与引用传递这两种传递参数的方式。这里给出一段实例代码,读者可以仔细领会这两种参数传递方式的异同。


```

/*参数的值传递与引用传递比较*/
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._2._3
{
    //定义 Program 类
    class Program
    {
        //整型参数 a 实现值传递, 整型引用参数 b 实现地址传递
        private void cal(int a, ref int b)
        {
            //对参数 a 进行运算
            a = a * a * a;
            Console.WriteLine("cal 中 a 的值是:{0}", a);
            Console.ReadLine();
            //对引用类型的参数 b 进行赋值
            b=a;
            Console.WriteLine("cal 中 b 的值是:{0}", a);
            Console.ReadLine();
        }
        public static void Main()
        {
            int a = 10;
            int b = 0;
            Program test = new Program();
            test.cal(a, ref b);
            //打印输出 a、b 的值进行比较
            Console.WriteLine("a 的值是:{0}", a);
            Console.WriteLine("b 的值是:{0}", b);
            Console.ReadLine();
        }
    }
}

```

程序编译后的结果是:

```

cal 中 a 的值是:1000
cal 中 b 的值是:1000
a 的值是:10
b 的值是:1000

```

这段示例程序很好地诠释了值传递与引用传递的不同。进入主程序 Main()以后, 首先定义并初始化了两个整型变量 int a=10, b=0。然后为类 Test 创建了一个新实例 test。调用 test 的 cal()方法, 并把变量 a 和 b 通过不同的方式传递进类 Test 的函数中。此时输出 cal 中 a 的值是 $10 \times 10 \times 10 = 1000$, b 的值是 1000。之后程序回到主函数中继续执行下一条语句。由于 a 只是传递了一个变量的复制, 即值传递时函数引用的是变量值的副本, 所以在主函数中 a 的值是不变的。变量 a 本身的值也没有发生改变。而引用传递中函数传递的值是对象地址的引用, 不传递具体值, 所以 b 的引用地址已经变成了最新的 1000 的地址而不再是原来的 0 的地址, 因此接下来的输出结果 a 的值是 10, b 的值是 1000。

⚠注意：在引用传递中，字符串是个例外。在 C# 中，不允许程序员通过引用类型而改变字符串变量的值。要改变字符串的值，只能重新创建一个新的字符串对象。

6.2.4 输出参数的使用

输出参数关键字为 `out`，形如 `function (out 形式参数)`。它也是一种引用传递方式，这种参数类型在函数超过一个以上的返回值时使用。使用 `out` 关键字一般是为了让一个方法有多个返回值。

示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._2._4
{
    //定义 Program 类
    class Program
    {
        //定义 Arr() 方法
        static public void Arr(out int[] arr)
        {
            //初始化数组
            arr = new int[8] { 111, 112, 113, 114, 116, 116, 117, 118 };
        }
        static void Main(string[] args)
        {
            //此处不必初始化
            int[] arr;
            Program.Arr(out arr);
            //输出 arr 数组中的数据
            for (int i = 0; i < arr.Length; i++)
            {
                Console.WriteLine(arr[i]);
                Console.ReadLine();
            }
        }
    }
}
```

执行编译以后，这段代码的输出结果为：

```
111
112
113
114
116
116
117
118
```

通过以上代码的比较可以知道，`ref` 与 `out` 传递的都是参数的地址。而它们的区别在于，数组类型的 `ref` 参数必须由调用方明确赋值，而使用数组类型的 `out` 参数前必须由定义函数


先为其赋值。

ref 有些类似于 C 语言中的指针。**ref** 参数一般用于传递数组参数，在调用之前变量一定要赋值，而在被调用函数内对应 **ref** 参数的变量可以修改，并且可以一起保存这种修改的结果。但不允许传递 **ref string**。

out 参数在调用时可以不初始化，即不用赋值，这与 **ref** 在使用前必须明确赋值是有差异的。而在被调函数内，**out** 参数引入的变量必须被至少赋值一次。**out** 关键字使参数在传递变量时，不必对函数体外的变量进行初始化就可以使用这些变量。一个函数中可以有一个以上的 **out** 参数。但是必须注意，属性不能作为 **out** 参数传递。

6.2.5 数组参数的使用

params 类型参数主要用于不知道数组长度的情况下进行函数声明。数组参数只能是一维的。在 C# 中，如果一个函数方法有多个输入参数，那么只允许函数携带一个 **params** 参数，而且 **params** 关键字修饰的这个参数必须是参数表中的最后一个。


 **说明：**数组参数的关键字为 **params**，形如 **function (params 形式参数)**，主要负责传递数组。

示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._2._5
{
    //定义 Program 类
    class Program
    {
        //定义 Add() 方法
        public static void Add(params int[] args)
        {
            int Count = 0;
            //args 为 8 维数组 {1, 2, 3, 4, 6, 6, 7, 8}
            foreach (int a in args)
            {
                Count += a;
            }
            Console.WriteLine("{0}", Count);
            Console.ReadLine();
        }
        static void Main(string[] args)
        {
            //声明并且定义整型 [8] 维数组 arr[8]
            int[] arr = new int[8] { 1, 2, 3, 4, 6, 6, 7, 8 };
            Add(arr);
        }
    }
}
```

这段代码的输出结果为：

在调用函数的时候，如果是有参数的函数，那么必然会涉及传递数据。能进行数据传递的函数的名称与参数类型必须匹配，参数数量一般也相等，当然某些库函数允许省略后续参数。

 **说明：**一般情况下，实际参数与形式参数的定义类型必须完全一致的时候才能进行数据的传递。不过在一些特殊的情况下，也可以进行调用函数并传递参数数值，这种情况实际上是一种强制类型转换。参加转换的两种类型必须是可以适应的，比如在函数中将形参定义为 int 型，将实参定义为 double 型，编译器在实际处理时将会把变量按 float 型处理。这是从 C 语言中继承而来的特性。

6.2.6 命名参数和可选参数

在 C# 4 中引入了命名实参和可选实参。利用“命名实参”，能够为特定形参指定实参，方法是将实参与该形参的名称关联，而不是与形参在形参列表中的位置关联。利用“可选实参”，能够为某些形参省略实参。这两种技术都可与方法、索引器、构造函数和委托一起使用。

在使用命名实参和可选实参时，将按实参出现在实参列表（而不是形参列表）中的顺序计算这些实参。

命名形参和可选形参一起使用时，能够只为可选形参列表中的少数形参提供实参。此功能大大方便了对 COM 接口（例如 Microsoft Office 自动化 API）的调用。

1. 命名参数

有了命名实参，不再需要记住或查找形参在所调用方法的形参列表中的顺序。可以按形参名称指定每个实参的形参。

例如，假设有一个根据人的身高、体重计算其胖瘦程序的函数：

```
CalculateBMI (int height, int weight)
```

通常按以下方式调用：

```
CalculateBMI (175, 70)
```

这时表示身高 175cm，体重 70kg。如果参数交换一下位置：

```
CalculateBMI (70, 175)
```

则会得到错误的计算结果。

这时，使用命名参数，则可防止出错。以下几种调用方式都会得到正确的结果：

```
CalculateBMI (height:175, weight:70)  
CalculateBMI (weight:70, height:175)
```

从上面的例子可看出，通过命名参数可解决参数的传入顺序问题。

2. 可选参数

所谓可选参数，是指在调用方法时可以省略的参数。每个可选形参都具有默认值作为其定义的一部分。如果没有为该形参发送实参，则使用默认值。默认值必须为常量。

可选形参在形参列表的末尾定义，位于任何必需的形参之后。

如果调用方为一系列可选形参中的任意一个形参提供了实参，则它必须为前面的所有可选形参提供实参。实参列表中不支持使用逗号分隔的间隔。

例如，在以下 `ExampleMethod` 方法中，使用一个必选形参和两个可选形参。

```
public void ExampleMethod(int required,
    string optionalstr = "default string",int optionalint = 10)
```

调用以上方法时可用以下方式：

```
Test.ExampleMethod(1,2,3);
Test.ExampleMethod(1,2);
Test.ExampleMethod(1);
Test.ExampleMethod(1, optionalint:3);
```

以上代码中第 1 行传入了 3 个参数，与通常情况下的调用方法相同，传入了所有参数。第 2 行调用时只传入了两个参数，第 3 个参数 `optionalint` 将使用默认值 10。第 3 行调用时只传入了 1 个参数，则第 2 个参数 `optionalstr` 和第 3 个参数 `optionalint` 将使用默认值。第 4 行调用时传入了两个参数，其中第 1 个参数按位置传入，第 2 个参数使用了命名参数，给第 3 个参数 `optionalint` 传入值，而第 2 个参数 `optionalstr` 则使用默认值。

下面的实例演示了命名参数和可选参数的使用方法。

```
namespace _6._2._6
{
    class Program
    {
        //具有命名参数和可选参数的方法
        public static void Search(string name, int age = 21,
            string city = "Pueblo")
        {
            Console.WriteLine("Name = {0} // Age = {1} // City = {2}",
                name, age, city);
        }
        static void Main(string[] args)
        {
            //标准调用
            Search("Sue", 22, "New York");
            //省略 city 参数
            Search("Mark", 23);

            //显式命名 city 参数
            Search("Lucy", city: "Cairo");

            //以相反顺序使用命名参数
            Search("Pedro", age: 45, city: "Saigon");
            Console.ReadLine();
        }
    }
}
```

6.2.7 利用返回值获取数值

在第5章中已经介绍过 `return` 语句的作用以及意义。本章中介绍的返回值基本上需要靠 `return` 语句的功能来实现。然而作为函数的一个组成部分，它的实际功能不仅仅局限于返回一个数值，更多时候需要的是一种判断结果，或者是某个标识信息。有了这种返回信息以后，程序才可以给其他部分或外部调用的其他程序传递必要的回馈，以进行下一步的控制操作。

返回数值的示例代码如下：

```
/*利用 return 返回值，确定空间点的位置*/  
// 声明使用的命名空间  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
namespace _6._2._7  
{  
    //定义 Point 类  
    class Point  
    {  
        //有参构造函数 Point(int x,int y,int z)，函数名和类名相同  
        public Point(int x, int y, int z)  
        {  
            this.x = x;  
            this.y = y;  
            this.z = z;  
        }  
        //定义辅助函数，确定空间点 (x,y,z) 的位置坐标  
        public int P_X( )  
        {  
            return (x);  
        }  
        public int P_Y( )  
        {  
            return (y);  
        }  
        public int P_Z( )  
        {  
            return (z);  
        }  
        //定义坐标变量  
        int x;  
        int y;  
        int z;  
    }  
    //定义 Program 类  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Point point = new Point(101, 202, 303);  
            //调用 point 对象的方法  
            Console.WriteLine("X 坐标= {0}", point.P_X( ));  
            Console.WriteLine("Y 坐标= {0}", point.P_Y( ));  
        }  
    }  
}
```



```

        Console.WriteLine("Z 坐标= {0}", point.P_Z());
        Console.WriteLine("空间点 point 坐标为({0},{1},{2})",
            point.P_X(), point.P_Y(), point.P_Z());
        Console.ReadLine();
    }
}

```

这一段代码主要实现了确定空间点坐标的功能。首先定义了一个空间点类 `Point`，在这个类中，利用构造函数实现了参数的传递。接下来在 `Point` 类中，通过定义辅助函数 `public int P_X()`、`public int P_Y()`，以及 `public int P_Z()` 来获得实例中的坐标值。然后在类中声明了坐标的 3 个维度变量 `x`、`y`、`z`。

最后在类 `Test` 中 `new` 了一个 `Point` 类的实例 `point`，并且初始化了这个坐标点的各个维度值，分别对应数组 `{1,2,3}`。在初始化的时候，编译器可以调用类 `Point` 中的方法函数，实现了对类中对象的赋值操作。下面的输出结果正好可以验证这一点。


输出结果为：

```

X 坐标= 101
Y 坐标= 202
Z 坐标= 303
空间点 point 坐标为{101, 202, 303}

```

通常情况下，一个函数只能返回一个返回值，而返回值主要由 `return` 语句来传递。当函数前面用 `void` 关键字来修饰的时候，没有返回值。

 **技巧：**作为一门面向对象的语言，C#语言可以将其他需要返回的类型封装在一个类中，然后返回这个类名就可以实现多返回值。

6.2.8 利用返回值判断逻辑

函数返回值除了被应用于需要有数值回馈的场合之外，另一个重要的使用类别就是返回判断值的真假了。示例代码如下：

```

/*利用 return 返回值，判断两数是否可以相除*/
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._2._8
{
    //定义 Class1 类
    class Class1
    {
        //定义 count() 方法
        public bool count(int a, int b)
        {
            if (b == 0)
                return false;
            else
                return true;
        }
    }
}

```

```

    }
}
//定义 Program 类
class Program
{
    static void Main(string[] args)
    {
        //定义变量
        int a, b;
        //输出提示信息, 并获得输入数值
        Console.WriteLine("请输入整数型除数");
        a = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("请输入整数型被除数");
        b = Convert.ToInt32(Console.ReadLine());
        // 创建 Class1 类的对象
        Class1 cal = new Class1();
        bool divide = cal.count(a, b);
        if (divide == true)
        {
            Console.WriteLine("除数不为零, 两数能够相除");
            float c = a / b;
            Console.WriteLine("两数相除的商是:{0}", c);
            Console.ReadLine();
        }
        else
        {
            Console.WriteLine("除数为零, 两数不能够相除");
            Console.ReadLine();
        }
    }
}
}

```

程序从主入口函数进入, 先定义了两个整型变量 `a` 和 `b`, 然后控制台输出提示语句“请输入整数型除数”。由于 `Console.ReadLine()` 函数的返回值是字符串类型的, 就是说用户从前台输入的整数实质上是 `string` 字符串, 因此需要添加一个强制转换函数 `Convert.ToInt32()`。同理, 输入除数以及被除数之后, 程序 `new` 了一个 `Class1` 类的实例, 然后调用类 `Class1` 并把从控制台输入的整数变量通过参数传入类中的 `count()` 方法函数。在 `Class1` 类中主要定义了一个布尔型函数 `count(int a, int b)`, 这个函数的主要作用是判断两个数当中的被除数是否为 0。当函数的一个参数 `b=0` 的时候返回 `false`, 返回值变成 `true`。

这段代码的执行结果如下:

```

请输入整数型除数
123
请输入整数型被除数
98
除数不为零, 两数能够相除
两数相除的商是:1.2661020

```

在本段代码中, 利用返回值实现了多次判断。这些返回值实际是布尔型变量的判断结果。通过这些 `return` 可以获得想要的某种程序指令用以执行之后的操作。

6.3 变量的作用域

C#的变量仅能从代码的本地作用域访问，因而程序员在定义变量的时候需要有一个作用域，对变量的访问实质上要通过这个作用域来实现。

6.3.1 最常见的局部变量

局部变量也被翻译成内部变量。在C#语言中，一对大括号“{}”中间的部分就是一个代码块，代码块决定其中定义变量的作用域。作用域仅覆盖一个函数的变量称为局部变量。还有一种全局变量，其作用域可覆盖几个函数。示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._3._1
{
    //定义 Program 类
    class Program
    {
        //自定义函数 Str()
        static void Str()
        {
            Console.WriteLine("String = {0}", str);
        }
        static void Main(string[] args)
        {
            //定义字符型变量 str
            string str = "String is defined in Main()";
            //调用函数 Str()
            this.Str();
            Console.ReadKey();
        }
    }
}
```

读者如果试着执行这段代码将会报错。这是因为变量 `str` 是定义在 `Main()` 函数中的，这就意味着 `str` 的作用域仅仅在应用程序主体 `Main()` 函数中才能有效，这也就证明了变量是有一个作用域的。变量的作用域包括定义变量的代码块和直接嵌套在其中的代码块。而自定义函数 `Str()` 是调用变量的代码块，这就超出了变量 `str` 的作用域范围。

可以将上面的程序修改如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._3._1
{
```

```
// 定义 Program 类
class Program
{
    // 定义 Str() 方法
    static void Str( )
    {
        string str = "String 在 Str() 函数中定义";
        Console.WriteLine("现在位于 Str() 函数中");
        Console.WriteLine("String = {0}", str);
    }
    static void Main(string[] args)
    {
        string str = "String 在 Main() 函数中定义";
        Str( );
        Console.WriteLine("现在位于 Main() 函数中");
        Console.WriteLine("String = {0}", str);
        Console.Read();
    }
}
```

这段代码是可以通过编译的。修改函数的定义位置实际上是修改了它们的作用域。这段代码执行的操作如下：

Main() 定义和初始化字符串变量 str，Main() 把控制权传送给自定义函数 Str()。在 Str() 函数中定义并且初始化一个字符串变量 str，它与 Main() 中定义的 str 变量完全不同。函数 Str() 把一个字符串输出到控制台上，该字符串包含在 Str() 中定义的 str 的值，之后 Str() 函数再把控制权传送回主控制函数 Main()。Main() 函数把一个字符串输出到控制台上，该字符串包含在 Main() 中定义的 str 的值。


这个示例程序的输出结果为：

```
现在位于 Str() 函数中
String = String 在 Str() 函数中定义

现在位于 Main() 函数中
String = String 在 Main() 函数中定义
```

6.3.2 需要慎用的全局变量

对于大多数的计算机语言来说，除了局部变量之外，还有另一种称为全局变量的变化，其作用域可覆盖几个函数。通常希望一个函数只能用于实现一个目的，这时，使用全局变量存储就能减少在函数调用中出错的可能性。

 **说明：**从严格定义上来讲，C# 语言中没有像 C 语言一样的真正的全局变量。在一个项目中声明的全局变量可以被涵盖在命名空间 namespace 中，而一个类当中所谓的全局变量则必须要用 static 关键字来修饰。

示例代码如下：

```
// 声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
```



```

using System.Text;
namespace _6._3._2
{
    //定义 Program 类
    class Program
    {
        //定义全局变量 str
        static string str;
        // 定义 Str() 方法
        static void Str( )
        {
            string str = " Str()中定义的字符串变量";
            Console.WriteLine("现在位于 Str()函数中");
            Console.WriteLine("局部变量 str = {0}", str);
            Console.WriteLine("全局变量 str = {0}", Program.str);
        }
        static void Main(string[] args)
        {
            string str = " Main()中定义的字符串变量";
            Program.str = "全局变量";
            Str( );
            Console.WriteLine("\n 现在位于 Main()函数中");
            Console.WriteLine("局部变量 str = {0}", str);
            Console.WriteLine("全局变量 str = {0}", Program.str);
            Console.Read();
        }
    }
}

```

程序输出结果如下：

```

现在位于 Str()函数中
局部变量 str = Str()中定义的字符串变量
全局变量 str =全局变量

现在位于 Main()函数中
局部变量 str = Main()中定义的字符串变量
全局变量 str =全局变量

```

通过以上程序可以知道，在编写控制台应用程序中，必须使用 `static` 或 `const` 关键字来定义全局变量。当在程序中需要修改全局变量的值时，就使用 `static` 关键字；当需要在程序中禁止修改变量的值时（相当于定义为全局常量），就使用 `const` 关键字。

为了区分全局变量和 `Main()` 与 `Str()` 中同名的局部变量，必须用一个完整限定的名称为变量名分类。在示例程序中，当全局变量和局部变量同名时把全局变量称为 `Program.str`。如果没有局部 `str` 变量，就可以使用 `str` 表示全局变量，而不需要使用 `Program.str`。假如任由局部变量和全局变量同名，而又没有使用一个完整限定的名称来访问全局变量，全局变量就会被屏蔽。

6.4 认识主入口函数 Main()

在 C# 中，每个程序都必须有一个主入口函数，即 `Main()` 函数。`Main` 的首字母必须大

写，且必须使用 `static` 关键字修饰。这也就是说，`Main()`函数必须是静态的，所以 `Main()`函数可以理解为一种比较特殊的静态方法。定义为 `static` 是为了把 `Main()`函数放在堆里，这样才能不用实例化该类就可以直接调用。

一个程序中仅能有一个名为 `Main()`的函数，该方法在类或结构的内部里声明。在 `Main()`函数中，如果调用此类中的非静态方法，必须先实例化这个类才能调用。但是可以直接调用此类中的任何静态方法。

声明 `Main()`函数时，既可以携带形式参数，又可以不使用参数。`Main()`函数不可以定义为公共方法，可以具有 `void` 或 `int` 返回类型。`Main()`函数有以下几种形态。

1. `static void Main()`

```
{  
    语句块  
}
```

示例代码如下：

```
//声明使用的命名空间  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
namespace _6._4._1  
{  
    //定义 Program 类  
    class Program  
    {  
        static void Main()  
        {  
            Console.WriteLine("static void Main() 示例程序");  
            Console.ReadLine();  
        }  
    }  
}
```

这是 C#语言中最简单的一段应用程序了。这段程序只有一个主入口函数 `Main()`，这个主函数本身不携带任何的参数。所以这段代码实现了一个最简单的语句输出命令。代码输出结果如下：

```
static void Main() 示例程序
```

在 C#的程序中，这种函数的利用率相对而言并不很高。而真正被广泛应用的是下面将要介绍的主入口函数类型。

2. `static void Main(string[] args)`

```
{  
    语句块  
}
```

`string[] args` 即命令行参数。同 C/C++语言一样，当启动 C#程序时，在命令行中可以输入参数，命令行参数指的就是在命令行中程序名后面输入的参数。`Main()`函数的传输参数存放在 `string` 数组 `args` 中。利用数组 `args` 的 `Length` 属性，可以得知命令行参数的个数。

示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._4._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("static void Main(string[] args) 示例程序");
            //重复命令行参数并且输出数值
            for (int arg = 0; arg < args.Length; arg++)
            {
                Console.WriteLine("Arg {0}: {1}", arg, args[arg]);
            }
            //从标准输入流下读取一段字符
            Console.ReadLine();
        }
    }
}
```


这段代码实现了 `static void Main(string[] args)` 函数的执行，并且为读者展示了 `Main(string[] args)` 函数中命令行参数的实际数值。程序除了实现了输出文字“static void Main(string[] args) 示例程序”之外，还输出了命令行参数的值。

程序的执行结果如下：

```
static void Main(string[] args) 示例程序
```

3. static int Main()

```
{
    语句块
}
```

 说明：Main() 函数的返回类型只能是 int 型。正常情况下会返回 0，表示调用成功；假如返回值为 1 则表示调用不成功。

示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._4._3
{
    //定义 Program 类
    class Program
    {
        public static int Main()
        {
```

```

        Console.WriteLine("static int Main() 示例函数");
        return (0);
    }
}

```

这段代码也是仅有一个主入口程序函数，然而与经常见到的情况略有不同的是，Main() 函数是有返回值的。返回的整数必须在 return() 函数中被传递。

这段代码的输出如下：

```
static int Main() 示例函数
```

主函数除了输出以上语句之外，同时返回了一个整数值 0。这个值并不输出，当其他程序或者项目中的其他函数，调用类 Program 中的这段函数时，可以接收到这个返回值 0 用以判断。

4. static int Main(string[] args)

```

{
    语句块
}

```

示例代码如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _6._4._4
{
    //定义 Program 类
    class Program
    {
        static int Main(string[] args)
        {
            Console.WriteLine("static int Main(string[] args) 示例程序");
            //重复命令行参数并且输出数值
            for (int arg = 0; arg < args.Length; arg++)
            {
                Console.WriteLine("Arg {0}: {1}", arg, args[arg]);
            }
            return (0);
        }
    }
}

```

这段代码的输出结果如下：

```
static int Main(string[] args) 示例程序
```


6.5 C#中最常用的函数

为了便于读者直观地理解并掌握 C#函数的定义与使用方法,现将所有常用函数的相关知识列表如表 6.1 所示。

表 6.1 常用函数

分 类	说 明	定 义
字符型	转为字符串	变量.ToString()
	转为 32 位数字型	Int32.Parse(变量)
		Int32.Parse("常量")
	字符串相加	System.Text.StringBuilder("")
字符型	截取字符串的一部分, 参数 1 为左起始位数, 参数 2 为截取几位	变量.Substring(参数 1, 参数 2)
	取字符串长度	变量.Length
	查指定位置是否为空字符	char.IsWhiteSpace(字符串变量, 位数)
	查字符是否是标点符号	char.IsPunctuation('字符串')
	把字符转为数字, 查代码表中的字符	(int)'字符'
	把数字转为字符, 查代码代表中的字符	(char)代码
	清除字符串前后空格	Trim()
	字符串替换	字符串变量.Replace("子字符串", "替换为")
	查找字符串中指定字符出现的位置并返回索引值	IndexOf()
	查找字符串中指定字符出现的位置并返回索引值	LastIndexOf()
	在字符串中指定索引位插入指定字符	Insert(,)
	在字符串左(或右)加空格或指定 char 字符, 使字符串达到指定长度	PadLeft(,), PadRight(,)
	从指定位置开始删除指定数的字符	Remove()
DateTime 数字型	取当前年月日时分秒	System.DateTime.Now
	取当前年	变量.Year
	取当前月	变量.Month
	取当前日	变量.Day
	取当前时	变量.Hour
	取当前分	变量.Minute

续表

分 类	说 明	定 义
DateTime 数字型	取当前秒	变量.Second
	取当前毫秒	变量.Millisecond
数字型	取 i 与 j 中的最大值	Math.Max(i,j)
字码转换	转为比特码	System.Text.Encoding.Default.GetBytes(变量)
网络	取远程用户 IP 地址	Request.ServerVariables["REMOTE_ADDR"]
	穿过代理服务器取远程用户真实 IP 地址	Request.ServerVariables["HTTP_X_FORWARDED_FOR"]
	Session["变量"]	Session["变量"]
	用超链接传送变量	String str=Request.QueryString["变量"]
XML	创建 XML 文档新节点	CreateElement("新建节点名")
	将新建的子节点加到 XML 文档父节点下	父节点.AppendChild(子节点)
	删除节点	父节点.RemoveChild(节点)
	跳转到 URL 指定的页面	Response.Redirect("URL 地址")
	向页面输出	Response.Write("字符串")
		Response.Write(变量)

这里列出的常用函数仅是 C# 支持的系统函数的一小部分, 如果读者需要应用某些特定函数去实现其功能时, 可以参考相关专业书籍, 本书在此就不详加讨论了。

6.6 本章总结

在本章中, 首先介绍了 C# 语言中的函数的常用表达形式和函数中的参数与返回值。然后介绍了局部变量和全局变量的作用与区别。C# 语言的函数主要分为无参函数和有参函数两种, 而函数的参数则可以是值参数、引用参数、输出参数和数组参数这四种。另外, 程序还可以通过函数的返回值获取数值或进行逻辑判断。

6.7 实战练习


1. 在 Visual Studio 2010 中新建一个控制台应用程序, 编写一个求最大值的函数 Max, 该函数对输入的若干个整数进行比较, 返回最大值。在 Main 函数中接收输入的数据, 并输出 Max 函数计算得到的结果。

 提示: 可定义一个数组来保存用户输入的数据, 然后将数组传递给 Max 函数。

2. 在 Visual Studio 2010 中新建一个控制台应用程序, 编写一个计算平均值的函数 Avg, 该函数对输入的若干个整数进行运算, 返回平均数。在 Main 函数中接收输入的数据, 并

输出 Avg 函数计算得到的结果。

3. 在 Visual Studio 2010 中新建一个控制台应用程序, 编写一个名为 Swap 的函数, 该函数的功能将传入的两个参数 x、y 的值进行互换, 既用 y 保存原来 x 的值, x 保存原来 y 的值。在 Main 函数中接收输入的数据, 并输出 Swap 函数互换后的结果。

提示: 定义函数时使用引用参数, 即可将交换的结果带回调用函数。

4. 在 Visual Studio 2010 中新建一个控制台应用程序, 编写一个名为 AgeRange 的函数, 该函数的功能是判断传入的参数是否在 18~60 之间, 若在此之间, 函数返回 true, 否则返回 false。在 Main 函数接收输入的一个整数, 然后将其传入 AgeRange 函数, Main 函数根据 AgeRange 函数的返回进行判断, 如果用户输入的年龄超过范围 (即 AgeRange 函数返回 false), 要求用户重新输入年龄数据。

第3篇 C#面向对象编程简介

- ▶▶ 第7章 类和对象
- ▶▶ 第8章 继承与多态
- ▶▶ 第9章 抽象类和接口
- ▶▶ 第10章 数组与集合
- ▶▶ 第11章 代理和事件


第7章 类和对象

在前面的章节中，已经对构成 C# 的基本数据类型进行了介绍，例如 int 型、long 型，以及 char 型。不过，C# 的精髓在于开发人员能够根据试图解决问题的不同，自己定义新的、复杂的，能够正确描述构成问题的数据类型。这种定义新类型的能力，正是面向对象语言所具有的重要特征。

本章将详细说明 C# 语言中关于类的一些重要特征。众所周知，构成类的成员的基本要素是其属性和方法（行为）。在本章中，将对用于定义类的行为和维护类中成员变量状态的方法进行说明。

7.1 类和对象的关系

在 C# 中，可以通过声明和定义类来实现特定类型的创建。此外，还可以将新的类型定义为接口，在下面的章节中会对此进行介绍。

 **说明：**一个类的实例将被称做对象，是程序执行过程中在内存中创建的。

类与对象之间的区别，是一种抽象和具体的区别。例如，书是一种抽象的概念，而读者正在阅读的这本书就是一个具体的存在。不过需要注意的是，只有在至少存在一个实例的情况下，才能够定义实例的抽象概念，不能凭空进行抽象。

以现实生活中的事物为例，一个定义为 Dog 的类描述了狗的如下特征，它的重量、身高、眼睛颜色、毛的颜色，以及脾气等。而且，还有它能够做的动作，例如吃东西、奔跑和睡觉等。而对于一条特定的狗（如邻居家的小狗），它就具有实例化的特征，如它有 3 斤重，20 厘米高，眼睛的颜色是黑色，毛的颜色是白色，并且脾气非常温顺等。同时，它能够完成所有狗都能做的基本动作。

在运用面向对象思想进行程序设计的过程中，使用类的最大好处在于它能将一个实体的属性和行为封装在一个独立的，并且能自我维护的代码单元中。例如，要对一个 Windows 列表框（List Box）实例的内容进行排序的话，可以直接调用该列表框实例内含的排序方法对其进行排序。列表框的使用者不用关注它是如何实现的，只需要了解它能够完成这一工作即可。这正是面向对象思想中封装原则的具体表现，封装、多态和继承是面向对象程序设计中 3 个重要的基本原则。

7.2 类的定义

7.2.1 创建一个类

在定义新的数据类型或者类的时候，需要先对数据类型或者类进行声明，再定义它们的方法和域（成员变量）。可以使用关键字 `class` 进行类的声明，完整的类的声明语法格式如下所示。

```
//类的声明格式
[attributes] [access-modifiers] class identifier [:base-class]
{
    class-body
}
```

其中，`attributes` 指明类的属性，对类的属性介绍将在后续章节中说明。`access modifiers` 定义类的访问类型，会在 7.2.2 节中进行说明。`identifier` 是所定义类的类名。可选项 `base-class` 是所定义类继承的基类，第 8 章将会对基类的概念详细说明。`class-body` 是由类的成员定义构成的，在成员定义前后需要用大括号将其括起来。

在 C# 中，所有事件都是在类中发生的，以如下的 `Sample` 类为例：


```
//声明 Sample 类
public class Sample
{
    public static int Main()
    {
        ..... //方法定义
    }
}
```

以上只是对 `Sample` 类的定义，而不是该类的实例。也就是说，还没有创建 `Sample` 类的对象。类与类的实例间的区别是什么呢？为了回答这个问题，先要理解 `int` 类型和 `int` 类型的变量之间的区别。通常，会进行如下定义：

```
int Temporary = 7;
```

而不能写成：

```
int = 7;
```

 **说明：**无论在什么情况下，都不可以把一个具体值赋给某种类型，而可以对类型的对象赋值。在上例中，变量 `Temporary` 就是 `int` 类型的对象。

在声明一个新类的时候，会对该类所有对象的属性进行定义，以及这些对象的行为（方法）。例如，在 Windows 程序设计中，如果想要创建一个窗口环境，并完成与用户的交互应用，通常要通过使用屏幕控件实现。其中，一种经常用于交互的控件就是列表框（`List box`），它能够为用户显示一系列当前可选的内容，并且可以让用户在列表中进行选择。

列表框控件具有多种特性，例如列表框的高（`height`）、宽（`width`）、屏幕位置（`location`）

和字体颜色 (text color)。程序设计人员还会希望列表框能具有其特定的行为, 如打开 (open)、关闭 (close) 和内容排序 (sort) 等。在面向对象程序设计中, 可以创建 ListBox 这样一个新的类型, 它将列表框的这些特性和行为封装在这个类型中。这样的一个新类型就是 ListBox 类, 它可以定义其成员变量, 分别是 height、width、location 和 text color, 也可以定义 sort()、add(), 以及 remove() 等成员方法。

在定义 ListBox 类型后, 是不能直接对它赋值的。而是首先要创建一个 ListBox 类型的对象, 如下述代码所示。

```
//创建 ListBox 类的对象
ListBox sampleListBox;
```

当创建 ListBox 的实例后, 就可以对实例的成员变量进行赋值了。

通过上述内容的介绍, 已经初步阐述了类和对象的概念。现在, 尝试定义一个类, 它能够记录并显示当天的时间。类的内部状态必须能够体现出当前的年、月、日, 以及小时、分钟和秒。同时, 还需要定义的类能够将时间以不同的格式显示。为了达到以上提出的这些要求, 定义的新类需要具有 6 个成员变量和 1 个成员方法, 如下代码所示。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._2._1
{
    //定义 Program 类
    class Program
    {
        //创建 Main() 函数
        static void Main(string[] args)
        {
            //创建 Student 类的对象
            Student t = new Student();
            //调用 Student 类对象 t 的成员方法
            t.Show();
        }
    }
    //定义 Student 类
    public class Student
    {
        //公共成员方法
        public void Show()
        {
            Console.WriteLine("显示学生信息!");
        }
        //私有变量
        string name;
        int ID;
        int mathsMark;
        int englishMark;
        int scienceMark;
        int totalMark;
    }
}
```




```

    }
}

```

在 `Student` 类的定义中只声明了一个成员方法，即 `Show()`，并且在定义类的过程中就定义了自己的方法过程。不像其他的面向对象语言（如 C++），C# 不必在方法定义之前先对其进行声明。而且 C# 语言不支持将类的声明和代码实现放在不同的文件中。所有 C# 的方法都可以向 `Student` 类中的 `Show()` 方法一样，在类的内部实现方法的定义。

 **注意：**C# 中没有头文件。

`Show()` 方法定义的返回类型是 `void` 类型，即调用该方法后不具有返回值。作为简单的示例，在方法定义过程中，以打印函数代替了方法的具体处理过程。

在 `Student` 类定义的一开始就对成员变量进行了声明，如 `name`、`ID`、`mathsMark`、`englishMark`、`scienceMark` 和 `totalMark`。

在 `Student` 类以大括号结尾之后，定义了两个 `Sample` 类。在 `Sample` 类中定义了熟悉的 `Main()` 方法，并且在 `Main()` 方法中创建了 `Student` 类的实例，用对象 `t` 表示新创建的实例。因为 `t` 是 `Student` 类的实例，所以 `Main()` 方法就可以利用 `Student` 类的对象 `t` 调用 `Show()` 方法，并用它显示学生信息。

7.2.2 类成员的访问类型

访问类型表明类中的哪些成员变量和方法对类中的其他方法（包括其他类中的方法）是可见的，以及可以调用的。如表 7.1 所示总结了 C# 中可用的访问类型。

表 7.1 C# 中的访问类型

访问类型	访问限定
Public	没有访问限制，对任何类的任何方法都是可见的
Private	类 A 中定义为 <code>private</code> 类型的成员只有类 A 中的方法才能进行访问
Protected	类 A 中定义为 <code>protected</code> 类型的成员只有类 A 中的方法，以及继承类 A 的类的方法才能进行访问
internal	类 A 中定义为 <code>internal</code> 类型的成员可以被类 A 所在程序集中的任何类的方法访问
protected internal	类 A 中定义为 <code>protected internal</code> 类型的成员可以被类 A 中的方法，以及继承类 A 的类的方法，或者类 A 所在程序集中的任何类的方法访问

通常情况下，都会把类的成员变量的访问类型定义为 `private` 类型。也就是说，只有该类的成员方法才能对这些变量进行访问和修改。因为在 C# 中 `private` 类型是默认的访问类型，所以在定义成员变量的时候不写出 `private` 也是可以的，但是为了程序的规范性还是建议明确指明访问类型。这样，前面示例代码所声明的成员变量应该规范的写成：

```

//私有变量
private string name;
private int ID;
private int mathsMark;
private int englishMark;
private int scienceMark;
private int totalMark;

```

说明：类 Sample 和方法 Show() 方法的访问类型是 public 类型，所以其他任何类都可以使用它们。

7.2.3 创建类的成员方法

类的成员方法可以有任意个参数，它们在方法名后的括号内定义，在定义的时候需要指定每个参数的类型和变量名称。下面的代码定义了 Method() 方法，它的返回值为 void（即没有返回值），并且有两个参数，分别是 int 类型和 string 类型。

```
//定义类的成员方法
void Method(int Param1, string Param2)
{
    //方法程序体
    .....
}
```

在方法的程序体内，参数可以当作内部变量使用，就像已经在程序体内进行了声明和初始化操作一样，其值就是方法参数的传入值。下面的示例代码说明如何定义一个方法，并将值传入方法内部。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._2._3
{
    //定义 Circle 类
    public class Circle
    {
        //定义 Circle 类的 Method() 方法
        public void Method(int firstParam, float secondParam)
        {
            //打印输出信息
            Console.WriteLine("接收到的参数分别是: {0}, {1}", firstParam, secondParam);
        }
    }
    //定义 Sample 类
    public class Sample
    {
        //定义程序的 Main() 方法
        static void Main()
        {
            //变量的定义和初始化
            int radius = 5;
            float pi = 3.14f;
            //创建 Circle 类的 test 对象
            Circle test = new Circle();
            //调用 test 对象的 Method() 方法
            test.Method(radius, pi);
        }
    }
}
```



```

    }
}

```

Method()方法有两个参数，分别为 int 类型和 float 类型，在程序体内通过使用 Console.WriteLine() 方法将参数输出到屏幕上。在 Method() 方法内部，firstParam 和 secondParam 两个参数被当成局部变量使用。

在 Main() 方法中，首先定义了 radius 和 pi 两个局部变量，并对其进行初始化操作。然后，将这两个变量作为参数传给 Method() 方法，编译器通过参数的位置进行值传递。

7.3 创建对象

C# 的数据类型中有两个大的类别，数值类型和引用类型。主要的 C# 数据类型（如 int 类型、char 类型等）都是数值类型，一般在栈上创建。而对象则是引用类型，通常使用关键字 new 在堆上创建，如下例所示。

```
Student t = new Student();
```

其中，t 没有实际包含 Student 类型对象的值，而是包含该对象的堆地址，t 本身只是 Student 类型对象的引用。


7.3.1 类的构造函数有什么用

在 7.2.3 节的例子中，创建 Student 对象的时候是通过调用一个方法进行的。而事实上，所有的类在实例化为对象的时候都要调用特定的方法。这个方法就是类的构造方法，该方法必须在定义类的时候作为一部分进行定义，或者由 Common Language Runtime (CLR) 自动提供。构造函数的作用就是创建特定类的对象，并赋给对象有效的状态值。在构造方法运行之前，对象就是一块未处理的存储区域。构造方法运行之后，这块存储区域就包含了一个类的有效的实例。

上述的 Student 类中，没有定义构造函数，而是由编译器提供的。默认的构造函数只负责创建对象，而不做其他的操作。类的成员变量都将初始化为不会引起运行错误的值，例如整数类型的默认值为 0，字符串类型的默认值为空字符串。如表 7.2 所示给出了 C# 对各种数据类型的初始化默认值：

表 7.2 不同数据类型的默认值

数据类型	默认值	数据类型	默认值
int	0	bool	False (假)
long	0	char	'\0' (空字符)
float	0.0	string	"" (空字符串)
double	0.0	Objects	null (空值)

 说明：通常情况下，用户都会定义自己的构造函数，并且根据构造函数参数设定的不同设置不同的对象初始化状态。这样，在使用类创建对象的时候，就可以使对象创建完成后就具备有意义的数据。

定义构造函数的时候，构造函数的函数名必须与所定义类的类名完全一致。构造函数没有返回值，而且一般情况下都被声明为 `public` 类型。如果构造函数需要使用参数，那么它的使用方法和类的其他方法一样。下例给出了 `Student` 类的构造函数，并且该构造函数具有 6 个参数，参数的类型是 `string` 型和 `int` 型，示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._3._1
{
    //定义 Student 类
    public class Student
    {
        //公共访问类型方法
        public void Show()
        {
            //打印输出学生信息
            System.Console.WriteLine("学生姓名: {0}", name);
            System.Console.WriteLine("学生学号: {0}", ID);
            System.Console.WriteLine("数学成绩: {0}", mathsMark);
            System.Console.WriteLine("英语成绩: {0}", englishMark);
            System.Console.WriteLine("自然成绩: {0}", scienceMark);
            System.Console.WriteLine("总成绩: {0}", totalMark);
        }
        //构造函数
        public Student(string st_name, int st_ID, int maths, int english, int science, int total)
        {
            //对类的私有成员变量进行赋值
            name = st_name;
            ID = st_ID;
            mathsMark = maths;
            englishMark = english;
            scienceMark = science;
            totalMark = total;
        }
        //私有成员变量
        string name;
        int ID;
        int mathsMark;
        int englishMark;
        int scienceMark;
        int totalMark;
    }
    //定义 Sample 类
    public class Sample
    {
        //定义程序的 Main() 方法
        static void Main()
        {
            //定义字符串变量 studentName
            string studentName = "小明";
            //定义 Student 类的对象 t，并调用 t 的成员方法 Show()
            Student t = new Student(studentName, 22, 90, 80, 90, 260);
            t.Show();
        }
    }
}
```



```

    }
}
}

```

程序执行后将输出学生的姓名和成绩等信息。本例中，构造函数有 6 个不同类型的参数，并在函数内部使用这些参数对成员变量进行赋值。当构造函数运行结束后，Student 类的对象就已经创建成功，并且对象的值已经被初始化。之后，当 Main() 方法中调用 Show() 方法后，就会将这些值显示出来。

在上述代码中，studentName 变量在 Sample 类的 Main() 方法中创建，并将其作为参数传给 Student 类的构造函数，其他 int 类型的参数直接使用数值常量。Student 类的对象在创建时，就使用构造函数传入的参数数值对类的私有成员变量进行初始化。

注意：在本例中使用的是数值进行参数传递，是在构造函数中创建数值的复制进行使用。如果使用对象作为参数的时候，传递的是对象的引用，而不是将对象复制一份传到方法体内。

7.3.2 成员变量的初始化

类的成员变量可以在定义的时候就对其进行初始化，而不用将每一个变量都在构造函数中执行初始化操作。成员变量的初始化如下所示。

```
private int totalMark = 300;
```

假设，Student 类的 totalMark 变量无论在何种情况下，它的初始化值都是 300。那么，就可以将 totalMark 变量的值在类定义的时候就进行初始化赋值，这样在构造函数中就可以不再对 totalMark 进行赋值。但是，如果有需要，还是可以通过构造函数修改其初值。如下例代码所示。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._3._2
{
    //定义 Student 类
    public class Student
    {
        //公共访问类型方法
        public void Show()
        {
            //打印输出学生信息
            System.Console.WriteLine("学生姓名: {0}", name);
            System.Console.WriteLine("学生年龄: {0}", ID);
            System.Console.WriteLine("数学成绩: {0}", mathsMark);
            System.Console.WriteLine("英语成绩: {0}", englishMark);
            System.Console.WriteLine("自然成绩: {0}", scienceMark);
            System.Console.WriteLine("总成绩: {0}", totalMark);
        }
    }
    //Student 类的构造函数
}

```

```

public Student(string st_name, int st_ID, int maths, int english, int
science, int total)
{
    //对 Student 类的私有成员变量进行赋值
    name = st_name;
    ID = st_ID;
    mathsMark = maths;
    englishMark = english;
    scienceMark = science;
    //totalMark 变量的显式赋值
    totalMark = total;
}
//Student 类的构造函数
public Student(string st_name, int st_ID, int maths, int english, int
science)
{
    //对 Student 类的私有成员变量进行赋值
    name = st_name;
    ID = st_ID;
    mathsMark = maths;
    englishMark = english;
    scienceMark = science;
}
//私有成员变量
string name;
int ID;
int mathsMark;
int englishMark;
int scienceMark;
//totalMark 变量的初始化赋值
int totalMark = 300;
}
//定义 Sample 类
public class Sample
{
    static void Main()
    {
        //定义字符串变量 studentName
        string studentName = "小明";
        //创建 Student 类的对象, 并调用对象的 Show() 成员方法
        Student t1 = new Student(studentName, 22, 90, 80, 90, 260);
        t1.Show();
        Student t2 = new Student(studentName, 22, 90, 80, 90);
        t2.Show();
    }
}
}

```

程序执行后的输出结果是:

```

学生姓名: 小明
学生学号: 22
数学成绩: 90
英语成绩: 80
自然成绩: 90
总成绩: 260
学生姓名: 小明

```




```

学生学号: 22
数学成绩: 90
英语成绩: 80
自然成绩: 90
总成绩: 300

```

在上述程序中, `totalMark` 成员变量在定义过程中进行了初始化赋值, 所以在创建 `t2` 对象的时候, 虽然没有使用构造函数对 `totalMark` 变量赋值, 但是通过打印结果得知 `totalMark` 的值为 300, 即通过初始化的方法对 `totalMark` 成员变量进行了赋值。不过, 作为对比, 程序中还创建了 `t1` 对象, 并且 `t1` 调用的构造函数对 `totalMark` 值进行了重写, 执行后发现 `t1` 对象的 `totalMark` 值已经不是 300, 而是参数小明的总成绩 260。由此可知, 在构造函数中可以对已经初始化的成员变量的值进行修改。

 **说明:** 如果对类的成员变量没有进行初始化赋值, 也没有在构造函数中进行赋值, 那么在对类进行实例化之后, 这些成员变量的值将是系统规定的默认值。

7.3.3 创建类的拷贝构造函数

拷贝构造函数也是类的构造函数的一种, 不过拷贝构造函数在对类的成员变量赋值时, 是通过将已创建的同类的其他对象的成员变量值复制给当前创建对象的变量实现的。例如, 如果想要创建一个与当前已存在的 `Student` 对象值相同的对象, 那么就可以使用拷贝构造函数将已存在的对象的值复制到新创建的对象中。

C#中没有现成的拷贝构造函数, 所以如果程序员想使用拷贝构造函数, 就必须自行实现。其原理就是使用构造函数将已有对象的变量值复制到新创建的对象中。

仍然使用 `Student` 类为例, 其拷贝构造函数可以写成:

```

//定义 Student 类的拷贝构造函数
public Student(Student existStudentObject)
{
    //利用 existStudentObject 对象的成员对 Student 类的私有成员变量进行赋值
    name = existStudentObject.name;
    ID = existStudentObject.ID;
    mathsMark = existStudentObject.mathsMark;
    englishMark = existStudentObject.englishMark;
    scienceMark = existStudentObject.scienceMark;
    totalMark = existStudentObject.totalMark;
}

```

拷贝构造函数在使用中进行如下调用即可:

```
Student t2 = new Student(t1);
```

其中, `t1` 是已存在的 `Student` 类的对象, `t2` 是新创建的 `Student` 类的对象。如上操作之后, `t2` 和 `t1` 两个对象的成员变量具有相同的值。

7.3.4 用关键字 `this` 引用当前对象

关键字 `this` 是对当前对象的引用, 可以在类的非静态成员方法中使用。通过使用 `this`

引用，类的成员方法可以在执行过程中调用类中的其他方法和成员变量。

通常情况下，`this` 引用有两种用法。第一种是使用 `this` 引用限定对象的成员变量，以避免由于参数名与成员变量名相似而引起执行错误。如下例所示：


```
public void Method(string name)
{
    //使用 this 引用给对象的成员变量进行赋值
    this.name = name;
}
```

在上例中，`Method()`方法的参数 `name` 与类的成员变量有相同的变量名。这种情况下，使用 `this` 引用可以解决变量名使用含糊不清的问题，如 `this.name` 代表类的成员变量，而 `name` 则表示方法的参数。`this` 引用这种用法的好处是，如果对于一个变量选定了非常合适的变量名，那么这个变量名既可以用做方法参数名，也用做类的成员变量名。

`this` 引用的第二种用法就是在一个方法中，将当前对象作为参数传给其他方法。下面代码给出了这种用法的示例：

```
public void Method(OtherClass otherObject)
{
    //使用 this 作为方法的参数
    otherObject.otherMethod(this);
}
```

上例中涉及两个类，具有 `Method()`方法的类和 `otherClass` 类。在 `Method()`中调用了 `otherClass` 类的 `otherMethod()`方法，并且通过使用 `this` 引用将当前调用 `Method()`方法的对象作为参数传给 `otherMethod()`方法。

 注意：`this` 还有第三种用法，就是配合数组的索引使用。这种用法将在后续的章节中介绍。

7.4 静态成员的使用

7.4.1 什么是类的静态成员

类的属性和方法既可以是实例成员，也可以是静态成员。其中，实例成员是与类的实例相关的一种类型，而静态成员则是类的一部分。对类的静态成员的访问，是通过类在声明时的名称进行的。例如，有一个 `Button` 类和它的两个实例 `btnUpdate` 与 `btnDelete`，同时 `Button` 类有一个静态成员方法 `Method()`。如果要访问这个静态方法，可以如下调用：

```
Button.Method();
```


而不能用下面的调用方法：

```
btnUpdate.Method();
```

在 C# 中，通过对象调用静态成员变量或方法是非法的。如果在程序中出现这种情况，会引起编译器的错误，这点与 C++ 中的静态成员有较大差别。

一些面向对象语言对类的方法和全局方法进行区分，全局方法是在所有类的外部定义，并且可以被其他方法直接调用。在 C# 中没有全局方法的概念，只有类的方法。但是，可以通过定义类的静态方法达到全局方法的作用。

类的静态方法与全局方法相比，有一定的好处，就是在调用静态方法的时候需要类名指定其范围，这样就可以避免在使用全局方法时产生的全局命名空间方法名混淆的情况。这可以帮助使用者设计更为复杂的程序，而类名对于静态方法而言可以被看做是一个命名空间。

说明：类的静态方法与全局方法有很多的相似之处，使用者可以在没有类的对象的情况下直接对该类的静态方法进行调用。

7.4.2 静态方法怎样调用

Main()方法就是静态方法。静态方法没有 this 引用，因为它没有可引用的实例。

在静态方法中，不能直接访问类的非静态方法。例如，在 Main()方法中想要调用一个非静态方法，就必须先创建这个类的对象。如下例所示：

```
//声明使用的命名空间
using System;
//定义 Circle 类
public class Circle
{
    public void Method(int firstParam, float secondParam)
    {
        //打印输出信息
        Console.WriteLine("接收到的参数分别是: {0}, {1}", firstParam,
            secondParam);
    }
}
//定义 Sample 类
public class Sample
{
    static void Main()
    {
        //对变量进行定义和初始化
        int radius = 5;
        float pi = 3.14f;
        //创建 Circle 类的对象，并调用其成员方法
        Circle test = new Circle();
        test.Method(radius, pi);
    }
}
```

Method()方法是 Circle 类的一个非静态方法，在 Main()方法中想要调用这个方法，首先要创建 Circle 类的对象 test，然后通过该对象调用 Method()方法。

修改以上代码为如下形式，将 Circle 类的 Method()定义为静态方法，则可直接在 Main()方法中调用这个方法，而不需要创建 Circle 类，具体代码如下：

```
//声明使用的命名空间
using System;
//定义 Circle 类
```

```

public class Circle
{
    static void Method(int firstParam, float secondParam)
    {
        //打印输出信息
        Console.WriteLine("接收到的参数分别是: {0}, {1}", firstParam,
            secondParam);
    }
}
//定义 Sample 类
public class Sample
{
    static void Main()
    {
        //对变量进行定义和初始化
        int radius = 5;
        float pi = 3.14f;
        //直接调用 Circle 类的静态方法
        Circle.Method(radius, pi);
    }
}

```


7.4.3 静态构造函数的优势

如果在类中定义了静态构造函数，那么静态构造函数必须在该类的任何实例化操作之前进行调用。例如，可以在 Student 类中添加如下静态构造函数，代码设置如下：

```

static Student()
{
    School = "育新小学";
}

```

 **注意：**在如上静态构造函数之间没有规定它的访问类型（如 public 等）。事实上，对所有的静态构造方法都不可以规定其访问类型。

由于静态成员方法是类的静态成员，无法对非静态的成员变量进行访问，所以变量 School 也必须声明成静态成员变量：

```
private static string School;
```

虽然静态构造函数可以完成对静态成员变量的赋值，但其实可以不使用静态方法完成这一目的。初始化操作也可以完成这一目的，代码设置如下：

```
private static string School = "育新小学";
```

但是，静态构造函数方法也有其优势。使用静态构造函数可以完成初始化操作无法完成的设置工作，并且这些工作很可能在运行过程中只允许进行一次，从而对所有该类的对象都产生作用。

7.4.4 使用私有构造函数保护静态成员

在 C# 中，由于没有全局方法和变量，所以程序设计人员可能会构造一些类来保存程序

中使用的静态成员。不管这种设计的优劣，如果创建了这样的类，那么一定不希望对其进行实例化。为此，可以定义一个不做任何操作而且没有任何参数的构造函数，并且定义该构造函数的访问类型为 `private`。由于没有公共构造函数，所以该类也不可能进行实例化操作，从而达到保护静态成员的目的。

7.4.5 使用静态成员变量记录对象数量

静态成员变量通常用来保存当前类的实例化对象的数量，该用法的示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._4._5
{
    //定义 Dog 类
    public class Dog
    {
        //定义 Dog 类的构造函数
        public Dog()
        {
            //对静态成员变量进行操作
            number++;
        }
        //定义 Dog 类的公共成员方法，用于输出已创建对象的数量
        public static void DogSum()
        {
            Console.WriteLine("共饲养{0}只狗!", number);
        }
        //定义 Dog 类的静态私有成员变量
        private static int number = 0;
    }
    //定义 Sample 类
    public class Sample
    {
        static void Main()
        {
            //创建 Dog 类的对象，并调用其成员方法
            Dog.DogSum();
            Dog d1 = new Dog();
            Dog.DogSum();
            Dog d2 = new Dog();
            Dog.DogSum();
        }
    }
}
```

程序执行后将输出结果：


```
共饲养 0 只狗！
共饲养 1 只狗！
共饲养 2 只狗！
```

Dog 类是对所饲养的狗的一个抽象，其中静态成员变量 `number` 是用来记录 Dog 类实现的对象的数量，并且在定义时初始化其值为 0。需要注意的是，静态成员是类的组成成员，但是却不是类的对象的成员，所以编译器不允许在创建对象的时候对其初始化。因此，静态成员变量必须在定义时就执行初始化操作，使用中必须注意。在 Dog 类的构造函数中，实现对 `number` 变量的自加操作，所以每当创建一个新对象的时候，`number` 变量的值就会增加 1，从而达到记录对象数量的目的。

7.5 对象的销毁

C#具有垃圾回收机制，因此不需要显式地定义类的析构函数。如果在程序设计过程中使用了较难管理的资源，并且在资源使用之后需要对资源进行释放，C#中提供了 `Finalize()` 方法完成这项工作。该方法在对象销毁的时候，由垃圾回收器负责调用。

`Finalize()`方法只能够释放那些由所要销毁的对象单独使用的资源。需要注意的是，如果在类中使用了易于管理的资源或引用，那么就没有必要实现 `Finalize()`方法。使用 `Finalize()`方法的目的是释放不易管理的资源，因为使用 `Finalize()`方法需要一定的系统消耗。

 **技巧：**程序设计人员不可以直接调用对象的 `Finalize()`方法，C#的垃圾回收器会在销毁对象的时候自动进行调用。只有一种情况除外，就是在实现类的 `Finalize()`方法的时候，可以调用基类的 `Finalize()`方法。

7.5.1 C#的析构函数

从语法上，C#的析构函数与 C++的析构函数十分相似。但是，两者的处理过程却有很大的不同。C#中的析构函数可以声明如下：

```
~testClass() { }
```

不过，在 C#中使用这种语法结构，只是为了提供一种声明与基类相关的 `Finalize()`方法的简单途径。析构函数的实现如下：

```
//定义 testClass 类的析构函数
~testClass()
{
    //析构处理
}
```

上述析构函数还可以写成：

```
testClass.Finalize()
{
    //析构处理
    base.Finalize();
}
```

这两种代码设计方式实现的作用是一致的，但是在使用的时候推荐第二种方法。


7.5.2 用 Dispose()方法释放资源

直接对 `Finalize()` 方法进行调用是非法的，因为在对象销毁的时候会由垃圾回收器自动调用该方法。但是，如果在对象中占用了比较稀有的资源，例如文件句柄，那么就需要尽快关闭或者处置这些资源，而不能等待 `Finalize()` 方法进行处理。为了达到这样的目的，就需要在程序设计过程中实现 `IDisposable` 接口。`IDisposable` 接口要求在实现过程中定义 `Dispose()` 方法，用于处理（多为释放）那些重要的资源。调用 `Dispose()` 方法之后，可以不用等待 `Finalize()` 方法执行，就能对资源进行释放，示例代码如下：

```
public void Dispose()
{
    //进行资源释放
    //通知 GC 停止对 Finalize() 方法的调用
    GC.SuppressFinalize(this);
}
```

在有些程序设计过程中，也可以将 `Dispose()` 方法放到 `Finalize()` 方法中调用，如下所示。

```
public override void Finalize()
{
    Dispose();
    base.Finalize();
}
```

 **说明：**如果在类中定义了 `Dispose()` 方法，那么就需要阻止垃圾回收器在对象销毁过程中对 `Finalize()` 方法的调用。为此，可以调用 `GC.SuppressFinalize()` 静态方法达到这一目的，该方法的参数是对象的 `this` 引用。

7.5.3 用 using 声明作用范围

因为不能保证对象在销毁的时候一定会调用 `Dispose()` 方法，同时 `Finalize()` 方法也可能无法及时执行（垃圾回收器执行的不可控性），所以 C# 提供了 `using` 声明，从而保证 `Dispose()` 方法能尽早地被调用。通常的使用方法是，声明 `using` 所控制的对象，并规定其作用范围。当执行到 `using` 的范围结尾处，将自动调用声明对象的 `Dispose()` 方法，如下例所示。

```
class Sample
{
    public static void Main()
    {
        using (string st1 = new string("Arial"))
        {
            //对 st1 对象进行操作
        } //编译器调用 string 类的对象的 Dispose() 方法
        string st2 = new string("Courier");
        using (st2)
        {

```


```

        //对 st2 对象进行操作
    } //编译器调用 string 类的对象的 Dispose() 方法
}

```

在上述代码中，两次进行了 `string` 类的实例化操作（即创建对象）。第一次是在 `using` 声明内部创建了 `st1` 对象。当 `using` 声明执行结束的时候，编译器会自动调用 `st1` 对象的 `Dispose()` 方法，释放其占用的资源。

第二次 `string` 类的实例化操作是在 `using` 声明外部，创建了 `st2` 对象，并将对象传入 `using` 声明内部进行使用，同样，当 `using` 声明执行结束的时候，编译器会自动调用 `st2` 对象的 `Dispose()` 方法，释放其占用的资源。

 **注意：**使用 `using` 声明还可以解决程序在执行过程中由于发生不可预料的异常而带来的安全问题。这是因为无论何种情况下，在离开 `using` 声明时都会调用 `Dispose()` 方法，完成资源的释放工作。这种处理就像使用 `try-catch-finally` 结构一样，在异常发生之后回收系统资源，两者达到的效果是相同的。

7.6 参数传递

在默认情况下，方法的参数是通过值传递进行的。也就是说，一个对象作为参数传递给方法使用，那么在方法内部就会给这个参数对象建立一个临时的拷贝。当方法运行结束时，建立的临时拷贝就会被销毁。虽然通常情况下都可以使用对象或变量的值进行传递，但有时也会遇到需要利用对象的引用传递参数的情况。C# 提供了 `ref` 和 `out` 参数类型用来向方法传递对象的引用，同时还提供了 `params` 类型用来向方法传递可变个数的参数。关键字 `params` 将在后续的章节中进行介绍。

7.6.1 通过引用传递返回多个值

方法调用后只能有一个返回值，但是有时希望方法能够返回更多的数据，这时就需要使用引用传递参数。

仍以 `Student` 类为例，现在对该类添加一个 `GetInfo()` 方法，调用该方法可以获得 `name`，`ID` 和 `totalMark` 的值。由于方法只具有一个返回值，因此可以向方法传递参数后，在方法执行过程中对参数变量进行修改，从而通过参数变量值的变化来获得所要的信息。参考如下代码：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._6._1
{
    //定义 Student 类
    public class Student

```



```

{
    //公共访问类型方法
    public void Show()
    {
        //打印输出学生信息
        System.Console.WriteLine("学生姓名: {0}", name);
        System.Console.WriteLine("学生年龄: {0}", ID);
        System.Console.WriteLine("数学成绩: {0}", mathsMark);
        System.Console.WriteLine("英语成绩: {0}", englishMark);
        System.Console.WriteLine("自然成绩: {0}", scienceMark);
        System.Console.WriteLine("总成绩: {0}", totalMark);
    }
    //定义 Student 类的 GetID() 成员方法
    public int GetID()
    {
        return ID;
    }
    //定义 Student 类的 GetInfo () 成员方法
    public void GetInfo(int st_ID, int st_maths, int st_total)
    {
        st_ID = ID;
        st_maths = mathsMark;
        st_total = totalMark;
    }
    //Student 类的构造函数
    public Student(string st_name, int st_ID, int maths, int english, int science, int total)
    {
        //对 Student 类的私有成员变量进行赋值
        name = st_name;
        ID = st_ID;
        mathsMark = maths;
        englishMark = english;
        scienceMark = science;
        totalMark = total;
    }
    //私有成员变量
    string name;
    int ID;
    int mathsMark;
    int englishMark;
    int scienceMark;
    int totalMark;
}

//定义 Sample 类
public class Sample
{
    //定义程序的 Main() 方法
    static void Main()
    {
        //定义字符串变量
        string studentName = "小明";
        //创建 Student 类的对象 t
        Student t = new Student(studentName, 22, 90, 80, 90, 260);
        t.Show();
        //定义变量并进行初始化
        int theID = 0;
        int theMaths = 0;
    }
}

```

```

        int theTotalMark = 0;
        //调用对象 t 的 GetInfo() 方法
        t.GetInfo(theID, theMaths, theTotalMark);
        //输出学生信息
        System.Console.WriteLine("学生信息: {0},{1},{2}", theID, theMaths,
            theTotalMark);
    }
}

```

程序执行后的输出结果是：

```

学生姓名: 小明
学生学号: 22
数学成绩: 90
英语成绩: 80
自然成绩: 90
总成绩: 260
学生信息: 0, 0, 0

```

从输出结果可以看出，调用 `GetInfo()` 方法后的输出结果是“0, 0, 0”，没有得到希望的结果。也就是说，`GetInfo()` 方法没有成功返回 `ID`、`mathsMark` 和 `totalMark` 的值。问题出现在 `GetInfo()` 方法的参数上，方法的 3 个参数变量都是 `int` 型的，在 `GetInfo()` 执行过程中对参数变量进行了修改，但是当方法返回后参数变量的值并没有改变。这是因为 `int` 类型变量是数值类型的，它们传递给方法后，在方法内部创建了参数变量的拷贝，方法内部的修改是针对复制的修改，而不是对参数本身的修改。在这种情况下，只有通过参数传递值的引用才能实现期望的功能。

为了实现引用参数的传递，需要对上述程序进行两点修改。首先，修改 `GetInfo()` 方法的参数类型，将参数指定为 `ref` 引用类型，示例代码如下：

```

public void GetInfo(ref int st_ID, ref int st_maths, ref int st_total)
{
    st_ID = ID;
    st_maths = mathsMark;
    st_total = totalMark;
}

```

其次，修改调用 `GetInfo()` 方法时传入的参数类型：


```

t.GetInfo(ref theID, ref theMaths, ref theTotalMark);

```

如果在调用 `GetInfo()` 方法的时候，没有指定参数是引用类型。那么，在编译的时候，编译器会提示无法将 `int` 类型的变量转换为 `ref int` 类型变量的错误。

经过上述修改，程序就可以输出正确的当前时间了。通过声明这些参数是 `ref` 类型，就可以使程序在运行的时候进行引用传递。与数值传递不同的是，引用传递没有在方法里建立参数变量的拷贝，方法内的参数变量是方法外部传入变量的引用，两者指向同一个变量。

 **注意：** 当在 `GetInfo()` 方法内部改变参数变量值的时候，外部变量的值也随之更改。

7.6.2 用 out 类型参数返回值

C#中规定,变量在使用之前必须进行赋值。在 7.6.1 节的程序中,如果对 theID、theMaths 和 theTotalMark 这 3 个变量没有进行赋值,就作为参数传给 GetInfo()方法,编译过程中会产生错误信息。所以,在使用这 3 个变量之前先将其值赋为 0,示例代码如下:

```
int theID = 0;
int theMaths = 0;
int theTotalMark = 0;
t.GetInfo(ref theID, ref theMaths, ref theTotalMark);
```

如果不进行赋值,就会产生如下错误:

```
Use of unassigned local variable 'theID'
Use of unassigned local variable 'theMaths'
Use of unassigned local variable 'theTotalMark'
```

但是,这种情况也不是绝对的。C#提供了 out 类型的参数,该类型可以不要求一个引用类型的参数在使用之前必须进行赋值。例如,GetInfo()方法的参数并没有给方法传入任何有用的信息,而只是用来从方法中传出信息。这种情况下,将参数定义成 out 引用类型,就不用在使用前对参数赋初值。不过,在方法返回之前,必须对 out 类型的参数变量进行相关的赋值操作。这样,GetInfo()方法可以修改成:

```
public void GetInfo(out int st_ID, out int st_maths, out int st_total)
{
    st_ID = ID;
    st_maths = mathsMark;
    st_total = totalMark;
}
```

在 Main()方法中就可以进行如下调用:

```
int theID;
int theMaths;
int theTotalMark;
t.GetInfo(out theID, out theMaths, out theTotalMark);
```

总结 3 种类型的参数传递,数值类型的参数给方法传递的是具体的数值。ref 类型的参数传递给方法的是某一变量的引用,并且允许在方法内部对变量的值进行修改。out 类型的参数只是用来从方法中获取具体信息。以下代码中对这 3 种类型的参数都进行了使用,注意比较其使用的不同。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._6._2
{
    //定义 Student 类
    public class Student
    {
        //公共访问类型方法
        public void Show()
```

```
{
    //打印输出学生信息
    System.Console.WriteLine("学生姓名: {0}", name);
    System.Console.WriteLine("学生年龄: {0}", ID);
    System.Console.WriteLine("数学成绩: {0}", mathsMark);
    System.Console.WriteLine("英语成绩: {0}", englishMark);
    System.Console.WriteLine("自然成绩: {0}", scienceMark);
    System.Console.WriteLine("总成绩: {0}", totalMark);
}
//定义 GetID() 方法
public int GetID()
{
    return ID;
}
//定义 SetInfo() 方法
public void SetInfo(int st_ID, ref int st_Maths, out int st_total)
{
    //如果传入的 st_Maths 值大于 100, 则将 mathsMark 和 totalMark 的值都设
    //为 0
    if (st_Maths > 100)
    {
        mathsMark = 0;
        totalMark = 0;
    }
    //根据传入参数设置 ID 的值
    ID = st_ID;
    //传出 mathsMark 和 totalMark 的值
    st_Maths = mathsMark;
    st_total = totalMark;
}
//Student 类的构造函数
public Student(string st_name, int st_ID, int maths, int english, int
science, int total)
{
    //对类的私有成员变量进行赋值
    name = st_name;
    ID = st_ID;
    mathsMark = maths;
    englishMark = english;
    scienceMark = science;
    totalMark = total;
}
//私有成员变量
string name;
int ID;
int mathsMark;
int englishMark;
int scienceMark;
int totalMark;
}
//定义 Sample 类
public class Sample
{
    //定义程序的 Main() 方法
    static void Main()
    {
        string studentName = "小明";
        Student t = new Student(studentName, 22, 90, 80, 90, 260);
    }
}
```



```

        t.Show();
        //定义变量, 并进行初始化
        int theID = 28;
        int theMaths = 80;
        int theTotalMark;
        //调用 Student 类的 SetInfo() 方法
        t.SetInfo(theID, ref theMaths, out theTotalMark);
        System.Console.WriteLine("学生信息: {0}, {1}, {2}", t.GetID(),
            theMaths, theTotalMark);
        theMaths = 105;
        t.SetInfo(theID, ref theMaths, out theTotalMark);
        System.Console.WriteLine("学生信息: {0}, {1}, {2}", t.GetID(),
            theMaths, theTotalMark);
    }
}
}

```


运行后的输出结果是:

```

学生姓名: 小明
学生学号: 22
数学成绩: 90
英语成绩: 80
自然成绩: 90
总成绩: 260
学生信息: 28, 90, 260
学生信息: 28, 0, 0

```

程序中设计 SetInfo() 方法的作用就是要说明 3 种不同类型参数的区别, 在实际中很难遇到这种功能的 SetInfo() 方法。首先, theID 参数是一个值传递, 它的作用就是对成员变量 ID 进行赋值, 使用该参数的时候不存在返回值。其次, theMaths 参数是 ref 类型的引用传递, 它有两个作用: 一是如果 theMaths 的值小于等于 100, 那么它就从方法中获得学生成绩的 mathsMark 值, 并将该值从方法中传递出来; 二是如果 theMaths 的值大于 100, 那么就将对象成员变量 mathsMark 和 totalMark 的值修改成 0, 之后将修改后的 mathsMark 值从方法中传递出来。最后, theTotalMark 参数是 out 类型的引用传递, 它的作用只是从方法中返回学生成绩的 totalMark 值。

说明: 从程序中可以看到, theID 和 theMaths 值在使用前必须赋初值, 它们的值在方法中可能会使用。而 theTotalMark 值在使用之前则不必进行赋值, 因为 out 类型的参数只是用来从方法中返回数据。

7.7 重写方法和构造函数让类更宜用

在程序设计的过程中, 经常需要同一名字的方法实现多种不同的功能。这种需求最明显的体现就是构造函数, 它经常要实现不同类型的对象初始化操作。以之前的 Student 类为例, 它的构造函数有 6 个 string 和 int 类型的参数。在函数执行过程中, 它将根据参数的数值对新建 Student 类对象的 name、ID、mathsMark、englishMark、scienceMark 和 totalMark

成员变量进行赋值。**Student** 类只有一个构造函数，也就是说对新建的对象只能进行一种初始化操作。如果能有多个构造函数，在新建不同对象的时候能进行不同的初始化操作，将会带来更大的方便。方法重写就是以这个为目的设计的。

方法的标记是通过定义其方法名和参数列表实现的，并且通过方法标记进行不同方法的识别。就是说，两个不同的方法，它们在方法名或者参数列表上一定存在区别。方法名不同不用做出具体解释，参数列表的不同可以体现在参数个数或者参数类型的不同。例如，以下 3 个方法声明代表 3 种不同的方法，示例代码如下：

```
//声明不同的方法
void Method(int p1);
void Method(int p1, int p2);
void Method(int p1, string p2);
```

其中，前两种方法的不同在于参数个数的不同，后两种方法的不同在于第二个参数的类型不同。类里可以定义任意个方法，但是必须保证方法之间的标记互不相同。下例对 **Student** 类进行修改，设计了两个构造函数，其中一个有 6 个参数，另一个只有两个参数，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._7
{
    //定义 Student 类
    public class Student
    {
        //公共访问类型方法
        public void Show()
        {
            System.Console.WriteLine("学生姓名: {0}", name);
            System.Console.WriteLine("学生年龄: {0}", ID);
            System.Console.WriteLine("数学成绩: {0}", mathsMark);
            System.Console.WriteLine("英语成绩: {0}", englishMark);
            System.Console.WriteLine("自然成绩: {0}", scienceMark);
            System.Console.WriteLine("总成绩: {0}", totalMark);
        }
        //定义 Student 类构造函数
        public Student(string st_name, int st_ID, int maths, int english, int science, int total)
        {
            //对类的私有成员进行赋值
            name = st_name;
            ID = st_ID;
            mathsMark = maths;
            englishMark = english;
            scienceMark = science;
            totalMark = total;
        }
        public Student(string st_name, int st_ID)
        {
            name = st_name;
            ID = st_ID;
```




```

        mathsMark = 0;
        englishMark = 0;
        scienceMark = 0;
        totalMark = 0;
    }
    //私有成员变量
    string name;
    int ID;
    int mathsMark;
    int englishMark;
    int scienceMark;
    int totalMark;
}
//定义 Sample 类
public class Sample
{
    static void Main()
    {
        string studentName = "小明";
        Student t1 = new Student(studentName, 22, 90, 80, 90, 260);
        t1.Show();
        Student t2 = new Student(studentName, 25);
        t2.Show();
    }
}
}

```

通过以上程序可看出，在 `Student` 类中有两个构造函数。如果方法标记只对方法名称进行区别，那么编译器在执行的时候将不知道创建 `t1` 和 `t2` 时该调用哪个构造函数。所以，方法标记不仅要考虑方法名，还要考虑方法的参数，这样编译器就可以为 `t1` 和 `t2` 选择合适的构造函数。即 `t1` 是使用 6 个参数的构造函数，`t2` 是使用两个参数的构造函数。

 **技巧：**在进行方法重写的时候，必须要保证重写的方法与原方法具有不同的方法标记，如不同的变量名称、个数或者类型。但是，在重写方法的时候，如果两个方法之间只有返回值类型不同，则这种方法重写是错误的，编译器会发出错误信息。关于这点，读者需要在实际使用过程中予以重视。

7.8 用属性封装类的数据

7.8.1 类的属性定义

类的属性是对类中包含数据的封装，是 C# 面向对象思想中类的封装性的充分体现。类的使用者可以像访问类的公共成员变量一样，直接对类的属性进行访问。不过，这种直接访问其实是通过类的方法间接进行的。

通常，类的使用者希望直接对类内的数据进行访问，而不用进行方法调用。然而，类的设计者却希望可以将类的数据进行隐藏，不能对其直接进行访问，而是通过具体的方法获取或修改。类的属性恰好可以解决这一问题。

说明：C#中属性的使用有两个目的，一是为类的使用者提供简单的接口，属性在使用的过程中就像访问类的成员变量一样简单；二是通过方法访问类的数据，可以满足面向对象的要求实现数据的隐藏。

属性在定义的时候，需要定义属性的类型和名字，之后跟随一对大括号。大括号内定义属性的 get 和 set 访问器，get 访问器没有参数，set 访问器有一个隐含的参数 value。需要注意的是，属性在定义的时候也没有参数。示例程序如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._8._1
{
    //定义 Student 类
    public class Student
    {
        //公共访问类型方法
        public void Show()
        {
            //打印输出学生信息
            System.Console.WriteLine("学生姓名: {0}", name);
            System.Console.WriteLine("学生年龄: {0}", ID);
            System.Console.WriteLine("数学成绩: {0}", mathsMark);
            System.Console.WriteLine("英语成绩: {0}", englishMark);
            System.Console.WriteLine("自然成绩: {0}", scienceMark);
            System.Console.WriteLine("总成绩: {0}", totalMark);
        }
        //Student 类的构造函数
        public Student(string st_name, int st_ID, int maths, int english, int science, int total)
        {
            //对 Student 类的私有变量进行赋值
            name = st_name;
            ID = st_ID;
            mathsMark = maths;
            englishMark = english;
            scienceMark = science;
            totalMark = total;
        }
        //创建属性
        public int ID
        {
            //返回私有变量 Id 的值
            get
            {
                return ID;
            }
            //对私有变量 Id 进行赋值
            set
            {
                Id = value;
            }
        }
        //私有成员变量
```



```

        string name;
        int Id;
        int mathsMark;
        int englishMark;
        int scienceMark;
        int totalMark;
    }
    //定义 Sample 类
    public class Sample
    {
        //定义程序的 Main() 方法
        static void Main()
        {
            string studentName = "小明";
            Student t = new Student(studentName, 22, 90, 80, 90, 260);
            t.Show();
            //调用 student 类的属性, 获得私有变量 Id 的值
            int theID = t.ID;
            System.Console.WriteLine("\nID 的值是: {0}\n", theID);
            theID++;
            //调用 student 类的属性, 对私有变量 Id 进行赋值
            t.ID = theID;
            int tem = t.ID;
            System.Console.WriteLine("修改后, ID 的值是: {0}\n", tem);
        }
    }
}

```

运行后的输出结果是:

```

学生姓名: 小明
学生学号: 22
数学成绩: 90
英语成绩: 80
自然成绩: 90
总成绩: 260
ID 的值是: 22
修改后, ID 的值是: 23

```

如上述程序所示, ID 属性在定义的时候实现了 `get` 和 `set` 访问器。每个访问器都包含程序体, 实现对属性值的读取和修改。属性的值既有可能存在数据库中, 也有可能像上述程序中一样, 存在类的私有变量中。

7.8.2 用 `get` 访问器返回属性值

`get` 访问器的作用有些像返回特定类型数值的类的方法, 不过它返回的是属性的值, 如下所示, 属性 ID 的 `get` 访问器返回的是类的私有成员变量的值, 其类型是 `int` 型。

```

get
{
    return ID;
}

```

在使用属性的时候, 不需要显式地调用, `get` 访问器也能够被调用从而读取属性的值,

代码设置如下：

```
Student t = new Student(studentName, 22, 90, 80, 90, 260);
int theID = t.ID;
```

可以看出，Student 对象的 ID 属性自动将其数值读取出来赋给 theID 局部变量，这就是 get 访问器自动调用读取属性值的结果。

7.8.3 用 set 访问器设置属性值

set 访问器用于对属性值进行修改，其返回值为 void。在定义 set 访问器的时候，必须使用 value 关键字表示传入的参数，从而实现对属性的赋值。代码设置如下：

```
set
{
    ID = value;
}
```

上例中，使用 value 实现对类的私有变量的赋值。同样，也可以使用 value 对数据库中的值或者其他成员变量进行赋值。在对属性进行赋值的时候，set 访问器自动被调用。代码设置如下：

```
theID++;
t.ID = theID;
```

7.9 用 Readonly 类型变量为类设置不变的值

仍以 Student 类为例，在设计 Student 类的时候可能希望它能提供表示当前时间的公共静态变量，当对该变量赋值之后，就不能对其进行修改，从而提供一个不变的参考数值。首先进行如下定义：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _7._9
{
    //定义 Student 类
    public class Student
    {
        //公共访问类型方法
        public void Show()
        {
            //打印输出学生信息
            System.Console.WriteLine("学生姓名: {0}", name);
            System.Console.WriteLine("学生年龄: {0}", ID);
            System.Console.WriteLine("数学成绩: {0}", mathsMark);
            System.Console.WriteLine("英语成绩: {0}", englishMark);
            System.Console.WriteLine("自然成绩: {0}", scienceMark);
        }
    }
}
```



```
        System.Console.WriteLine("总成绩: {0}", totalMark);
    }
    //Student 类的构造函数
    public Student(string st_name, int st_ID, int maths, int english, int
science, int total)
    {
        //对 Student 类的私有变量进行赋值
        name = st_name;
        ID = st_ID;
        mathsMark = maths;
        englishMark = english;
        scienceMark = science;
        totalMark = total;
    }
    //创建属性
    public int ID
    {
        get
        {
            return Id;
        }
        set
        {
            Id = value;
        }
    }
    //私有成员变量
    public string name;
    public int Id;
    public int mathsMark;
    public int englishMark;
    public int scienceMark;
    public int totalMark;
}
//定义 JustThisStudent 类
public class JustThisStudent
{
    //定义 JustThisStudent 类的构造函数
    static JustThisStudent ()
    {
        string studentName = "小明";
        Student t = new Student(studentName, 22, 90, 80, 90, 260);
        name = t.name;
        ID = t.ID;
        mathsMark = t.mathsMark;
        englishMark = t.englishMark;
        scienceMark = t.scienceMark;
        totalMark = t.totalMark;
    }
    //私有成员变量
    public static string name;
    public static int ID;
    public static int mathsMark;
    public static int englishMark;
```

```

        public static int scienceMark;
        public static int totalMark;
    }
    //定义 Sample 类
    public class Sample
    {
        static void Main()
        {
            System.Console.WriteLine ("该学生 ID 是: {0}! ",
                JustThisStudent.ID);
            JustThisStudent.ID = 20;
            System.Console.WriteLine ("该学生 ID 是: {0}! ",
                JustThisStudent.ID);
        }
    }
}

```

程序执行后的输出结果是:

```

该学生 ID 是: 22!
该学生 ID 是: 20!

```

从输出结果可以知道, JustThisStudent.ID 的值在执行过程中被修改了, 明显没有实现不能进行修改的目的。如果将变量定义成 `constant` 类型的话, 可以实现无法对值进行修改的目的。但是, 由于类的成员变量是在构造函数执行的过程中进行赋值的, 而不是初始化赋值, 所以无法将变量定义为 `constant` 类型。

为了解决这一问题, C#提供了 `readonly` 关键字, 可以将变量定义成只读类型。这样变量在赋值之后, 就无法再对其进行修改。`readonly` 关键字的使用方法如下:

```

public static string name;
public static readonly int ID;
public static readonly int mathsMark;
public static readonly int englishMark;
public static readonly int scienceMark;
public static readonly int totalMark;

```

进行如上定义之后, 如果还想对 ID 变量的值进行修改, 编译器就会产生错误信息。

7.10 本章总结

传统的结构化程序设计是面向过程的, 虽然这种程序设计的方式得到了广泛的应用, 但是随着应用的不断深入, 结构化程序设计逐步体现并且暴露出其无法避免的缺点。面向对象程序设计思想的出现, 解决了结构化程序设计中的这些问题。面向对象的程序设计是对现实世界利用软件程序的直接模拟, 它将特定对象的数据和对数据的处理封装成了一个整体, 这个整体就是类。因此, 可以说类是整个面向对象程序设计的基础。

C#是一种面向对象程序设计语言, 所以必须对 C#中类的概念有明确的认识。本章对 C#中的类的基本知识进行了介绍, 并具体说明了类的各种组成。

C#中的类是对各种不同数据结构的抽象和封装,其中不仅包含抽象的数据内容,还包含对这些数据的不同操作。可以说类是C#的基础,利用类可以定义新的数据类型,从而满足程序设计的要求。类完成对数据和操作的封装后,就能够有效地控制类外部成员对这些成员的访问,提高数据使用的安全性。

在本章中介绍了类的定义和创建操作。类的定义是程序设计的基础。在类的定义过程中,明确了类所包含的数据、对数据的操作,以及数据和操作的访问类型等一系列信息。在完成类的定义之后,就可以创建类的对象。创建类的对象就是对该类的实例化操作,是由抽象到具体的一个过程。在类的实例化过程中,最先执行的代码就是类的构造函数,从而完成对将要创建对象的初始化操作。类的构造函数既可以根据用户的需求定义相应的初始化参数,也可以使用默认无参数的构造函数。

当类的实例(即对象)完成相关操作,或者超出作用域时,就可以使用类的析构函数来完成对象所占内存空间的释放操作。C#中具有垃圾回收机制,能够有效地管理无用对象的释放操作,但是在程序设计过程中还是需要对这方面加以关注。

关于静态成员,需要明白类的静态成员属于该类,而非静态成员则是属于这个类的某个实例。

在本章中还介绍了类的方法的参数传递和重写操作,以及类的属性等相关概念,这些知识在类的创建和使用中具有比较关键的作用,需要仔细地进行理解。

7.11 实战练习

1. 在 Visual Studio 2010 中新建控制台应用程序,编写一个名为 Circle 的类,在其中编写静态方法 Area,根据传入的圆半径参数,计算并返回圆的面积;编写静态方法 Perimeter,根据传入的圆半径参数,计算并返回圆的周长。

2. 在 Visual Studio 2010 中新建控制台应用程序,编写一个名为 Car 的类,通过属性 Type、Speed、Color 等分别表示车辆类型、速度、颜色等。在 Car 类中编写 Acceleration 方法,该方法根据传入的参数进行加速或减速,并输出现在的速度。再编写一个名为 Brake 方法,该方法可将速度设置为 0。编写好该类后,在 Main 中实例化 Car 类,并调用 Acceleration 方法和 Brake 方法进行加速和刹车操作。

第 8 章 继承与多态

继承和多态是面向对象程序设计的重要特性，本章将对 C# 中的继承和多态进行介绍。本章的内容安排如下：首先，会对继承机制的原理进行说明，并介绍 C# 中对继承的支持；之后，会研究 object 类的作用，以及多态在 C# 中的实现；最后，将对 C# 中类型的装箱和拆箱机制的工作原理进行说明。

8.1 用继承对类进行扩展

8.1.1 面向对象程序中的继承

凡是接触过面向对象程序设计的人员，都应该听说过面向对象的重用性和扩展性。重用性是指一种模型（可以是组件、类，甚至是方法）可以在几乎不需要对代码进行修改的情况下，被多种不同的应用程序使用的属性。模型的扩展性是指，在新的需求产生的时候，可以对其进行扩充和加强的特性。在面向对象程序设计中，重用性是通过减少不同类之间的耦合实现的，而扩展则是通过子类的应用实现的。这种通过创建一个类的子类，从而扩展其功能的过程被称为继承。

在面向对象的概念里，被扩展的原始类叫做基类、父类或者超类。而对基类的功能进行继承以及功能扩展的类，叫做子类、派生类或者继承类。其关系如图 8.1 所示。

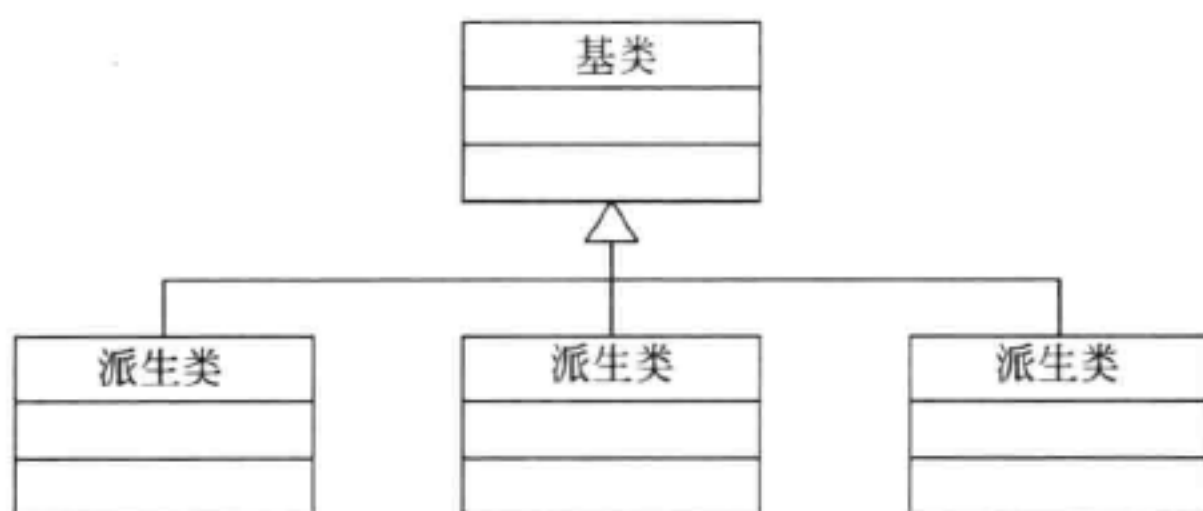


图 8.1 基类与派生类关系图

图 8.2 为继承实现的具体实例。图中使用统一模型语言（Unified Modeling Language, UML）的类图对继承的关系进行了说明。这里，Shape 是一个基类，而 Circle、Rectangle 和 Curve 则是 Shape 类的继承类。所以，在创建子类或进行继承的时候，又可以称为“由特殊到一般”的过程。

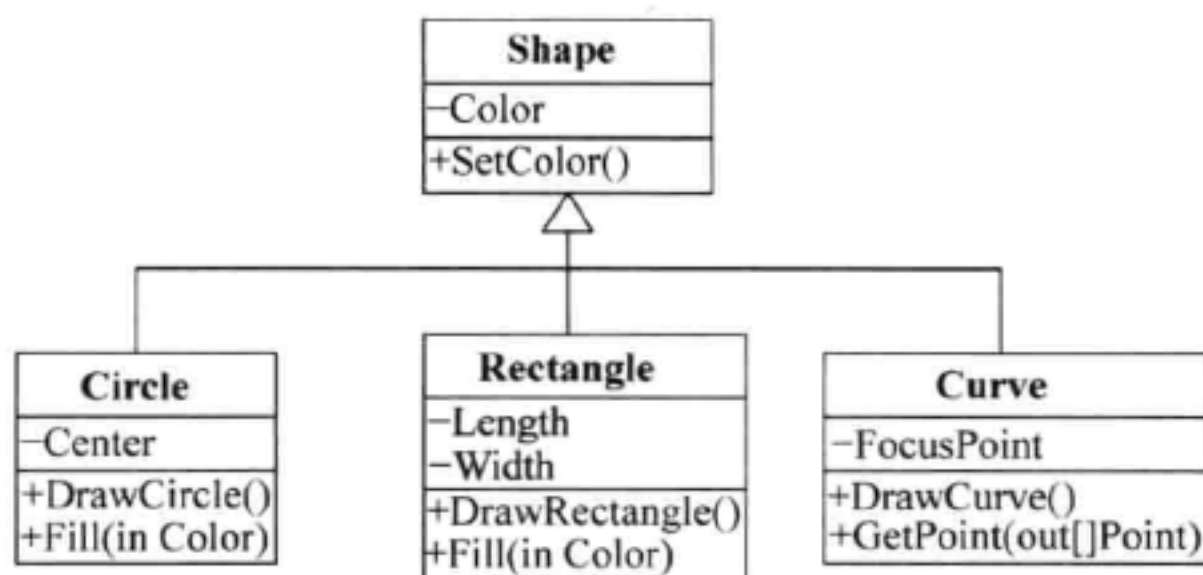


图 8.2 继承实现实例图

说明：关于统一模型语言的详细内容，请读者参阅有关文档。

如果子类 B 是对基类 A 的继承，那么 B 类将具有 A 类所有的特征和行为，即拥有 A 类的成员变量、方法和属性，而且 B 类又按照自身的特殊需求在 A 类的基础上进行了功能扩展。在实现继承之后，B 类可以对 A 类的非私有成员进行访问。这并不是说子类没有对基类的私有成员进行继承，只是无法在子类中进行访问。继承可以理解成子类是对基类的实例化，也就是说子类是基类类型的实例。

8.1.2 C#的继承机制

在对 C# 中的继承机制进行详细介绍之前，需要说明以下几点：

- (1) C# 只允许类进行单继承，不可以进行多重继承。这一点与 Java 类似，而与 C++ 正好相反；
- (2) 在 C# 中，定义在 System 命名空间里的 Object 类，是所有类的基类；
- (3) 与 Java 相似，C# 里的接口可以继承多个接口。

说明：C# 中使用冒号“:”作为表示继承的操作符。

为了对继承进行介绍，首先设计一个 Student 类。该类有 3 个成员变量，分别是 registrationNumber、name 和 dateOfBirth，以及相应的属性。同时，类中还有一个 GetAge() 方法，用来获得学生的年龄。类的定义如下：

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _8._1._2
{
    //定义 Student 类
    class Student
    {
        //私有成员变量
        private int registrationNumber;
        private string name;
    }
}
  
```

```
private DateTime dateOfBirth;
//公共访问类型方法
public int GetAge( )
{
    int age = DateTime.Now.Year -dateOfBirth.Year;
    return age;
}
static void Main()
{
}
//Student 类的构造函数
public Student( )
{
    //打印输出结果
    Console.WriteLine("创建新的 Student 类对象, 没有进行初始化。");
}
public Student(int registrationnumber, string name, DateTime
dateOfBirth)
{
    //对 Student 类的成员变量进行赋值
    this.registrationNumber = registrationnumber;
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    Console.WriteLine("创建新的 Student 类对象, 完成初始化。");
}
//Student 类的 RegistrationNumber 属性
public int RegistrationNumber
{
    get
    {
        return registrationNumber;
    }
}
//Student 类的 Name 属性
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
//Student 类的 DateOfBirth 属性
public DateTime DateOfBirth
{
    get
    {
        return dateOfBirth;
    }
}
```



```

    }
    set
    {
        dateOfBirth = value;
    }
}
}
}

```

Student 类的结构比较简单, 只有 3 个成员变量及其相应属性和一个用于获取学生年龄的成员方法。此外, Student 类有两个构造函数, 分别进行有参数和没有参数的对象创建。需要注意的是, Student 类的 RegistrationNumber 属性只定义了 get 访问器, 因为在对学生学号初始化之后, 不希望再对其进行修改。

现在, 定义一个继承 Student 类的 SchoolStudent 类, 并且在该类中增加一些变量和方法, 代码如下:

```

//定义继承 Student 类的 SchoolStudent 类
class SchoolStudent : Student
{
    //公共访问类型方法
    public double GetPercentage( )
    {
        percentage = (double) totalObtainedMarks / totalMarks * 100;
        return percentage;
    }
    //SchoolStudent 类的构造函数
    public SchoolStudent( )
    {
        Console.WriteLine("创建新的 SchoolStudent 类对象, 没有进行初始化。");
    }
    public SchoolStudent(int registrationNumber, string name, DateTime
        dateOfBirth,int totalMarks,
            int totalObtainedMarks)
        : base(registrationNmuber, name, dateOfBirth)
    {
        //对 SchoolStudent 类中的变量进行赋值
        this.totalMarks = totalMarks;
        this.totalObtainedMarks = totalObtainedMarks;
        Console.WriteLine("创建新的 SchoolStudent 类对象, 完成初始化。");
    }
    //定义 SchoolStudent 类的 TotalMarks 属性
    public int TotalMarks
    {
        get
        {
            return totalMarks;
        }
        set
        {
            totalMarks = value;
        }
    }
}

```


```

    }
    //定义 SchoolStudent 类的 TotalObtainedMaks 属性
    public int TotalObtainedMarks
    {
        get
        {
            return totalObtainedMarks;
        }
        set
        {
            totalObtainedMarks = value;
        }
    }
    //私有成员变量
    private int totalMarks;
    private int totalObtainedMarks;
    private double percentage;
}

```

SchoolStudent 类通过使用 “:” 操作符表示继承 Student 类，代码如下：

```
class SchoolStudent : Student
```

 注意：SchoolStudent 类不仅继承了 Student 类中所有的成员，而且还定义了自己的成员变量和方法。

在 SchoolStudent 类中增加了 3 个新的私有变量（totalMarks、totalObtainedMarks 和 percentage）及其相应属性、两个构造函数（有参数和没有参数）和一个成员方法（GetPercentage()）。下面对这两个类进行使用，代码如下：


```

class Sample
{
    static void Main(string [] args)
    {
        Student std = new Student(1, "李明", new DateTime(1984, 12, 19));
        Console.WriteLine("学生{0}的年龄是{1}\n.", std.Name, td.GetAge( ));
        //创建 SchoolStudent 类的对象
        SchoolStudent schStd = new SchoolStudent( );
        //利用 schStd 对象的属性进行赋值
        schStd.Name = "王华";
        schStd.DateOfBirth = new DateTime(1985, 4, 1);
        schStd.TotalMarks = 500;
        schStd.TotalObtainedMarks = 486;
        Console.WriteLine("学生{0}的年龄是{1}, 成绩百分比是{2}%.", schStd.Name,
            schStd.GetAge( ), schStd.GetPercentage( ));
    }
}

```

在 Main()方法中，首先创建了一个 Student 类的对象 std，并且将学生 std 的名字和年龄进行打印输出。然后，创建了 SchoolStudent 类的对象 schStd。在创建 schStd 的时候，没有使用带参数的构造函数，而是通过对象的属性对其进行赋值，并打印输出学生 schStd 的

名字、年龄和成绩百分比等信息。

说明：在对 schStd 对象赋值的时候，使用了 Student 类的 Name 和 DateOfBirth 属性，这是因为 SchoolStudent 类在继承 Student 类的同时，也继承了该类的公共属性。

程序在执行后的输出结果如下：

```
创建新的学生对象，完成初始化。
学生李明的年龄是 23。
创建新的学生对象，没有进行初始化。
创建新的 school 学生对象，没有进行初始化。
学生王华的年龄是 22，成绩百分比是 95.2%。
```

输出结果的前两行，实际是 S 在创建 std 对象和执行 GetAge()方法时输出的。但是在创建 schStd 对象的时候，却输出了如下的结果：

```
创建新的学生对象，没有进行初始化。
创建新的 school 学生对象，没有进行初始化。
```

这是由于继承类在执行构造函数的时候，将自动执行基类的构造函数造成的。

8.1.3 继承是怎样调用构造函数

如上一节中的程序所示，当继承类 SchoolStudent 进行实例化的时候，编译器将首先调用基类 Student 相应的构造函数，然后再调用继承类的构造函数。现在，假设将要调用 SchoolStudent 类的第 2 个（有参数的）构造函数，并且对该构造函数做一些简单的修改，代码如下：

```
public Student(int registrationNumber, string name, DateTime
dateOfBirth,int totalMarks, int totalObtainedMarks)
    //: base(registrationNmuber, name, dateOfBirth)
{
    this.Name = name;
    this.DateOfBirth = dateOfBirth;
    this.totalMarks = totalMarks;
    this.totalObtainedMarks = totalObtainedMarks;
    Console.WriteLine("创建新的 SchoolStudent 类对象，完成初始化。");
}
```

上述代码对原构造函数进行了修改，去掉了原代码的 base 关键字所在的代码行，而且修改了构造函数内的赋值过程，使用基类的属性，对基类的成员变量进行赋值。修改后的构造函数，可以在 Main()方法中进行如下调用：

```
static void Main(string [] args)
{
    SchoolStudent schStd = new SchoolStudent(2, "王华", new DateTime(1985,
4, 1), 500, 486);
    Console.WriteLine("学生{0}的年龄是{1}，成绩百分比是{2}%。",
        schStd.Name,schStd.GetAge( ),schStd.GetPercentage( ));
}
```

在 Main() 方法中, 使用带有参数的构造函数创建了一个 SchoolStudent 类的对象, 然后对学生的姓名、年龄和成绩百分比等信息进行了打印输出。代码运行后的输出结果如下:

```
创建新的学生对象, 没有进行初始化。
创建新的 school 学生对象, 完成初始化。
学生王华的年龄是 22, 成绩百分比是 95.2%。
```

分析构造函数的打印结果, Student 基类调用的是无参数的构造函数, 而 SchoolStudent 子类调用的是有参数的构造函数。可以看出, 在创建子类对象的时候, 编译器首先对基类进行实例化, 并且如果没有显式规定基类调用的构造函数时, 会自动调用基类无参数的构造函数。现在将 Student 类中无参数的构造函数注释后, 看看会发生什么情况。代码如下:

```
class Student
{
    ...
    /*public Student( )
    {
        Console.WriteLine("创建新的 Student 类对象, 没有进行初始化。");
    }*/
    ...
}
```

修改后, 对程序进行编译, 编译器会发出如下错误信息:

```
No overload for method 'Student' takes '0' arguments.
```

通过以上错误信息可以看出, 编译器无法在 Student 基类中找到匹配的无参数的构造函数。出现这种情况需要怎么解决呢? 一是可以在基类中定义无参数的构造函数, 二是使用 8.1.4 节中介绍的 base 关键字显式地规定基类使用的构造函数。

8.1.4 用 base 关键字调用基类构造函数

使用 base 关键字可以在继承类实例化的时候, 显式地定义应该调用的基类的构造函数。base 关键字必须在定义子类构造函数的时候与 “:” 操作符配合使用, 位置在构造函数的函数头后面。如下代码所示。

```
class SubClass : BaseClass
{
    //显式的指定基类使用的构造函数
    public SubClass(int parameter) : base( )
    {
        //构造函数处理过程
    }
}
```

在上述代码中, 显式地指明子类在实例化的时候将调用基类无参数的构造函数。当然, 也可以用 base 关键字指明调用基类有参数的构造函数。代码如下:

```
class SubClass : BaseClass
{
```



```

public SubClass(int parameter) : base(parameter)
{
    //构造函数处理过程
}

```

进行如上定义后,子类的构造函数在执行的时候,将会自动调用带有 `int` 型参数的基类的构造函数。这样就能知道,调用基类的哪个构造函数,是通过定义 `base` 的时候其后所跟的参数的类型以及个数决定的。基类中必须有符合参数类型和个数的构造函数,否则编译器会产生错误信息。

如在 `SchoolStudent` 类中,使用带有参数的构造函数,并通过 `base` 关键字指明调用的基类构造函数,代码如下:

```

public Student(int registrationNumber, string name, DateTime
dateOfBirth,int totalMarks, int totalObtainedMarks)
    : base(registrationNmuber, name, dateOfBirth)
{
    this.totalMarks = totalMarks;
    this.totalObtainedMarks = totalObtainedMarks;
    Console.WriteLine("创建新的 SchoolStudent 类对象,完成初始化。");
}

```

如上 `SchoolStudent` 类带有参数的构造函数,指明在执行的时候将调用 `Student` 基类的有参数的构造函数,对 `Student` 类的成员变量进行初始化赋值。调用的基类构造函数如下所示。

```

public Student(int registrationNumber, string name, DateTime dateOfBirth)
{
    this.retistrationNumber = registrationNumber;
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    Console.WriteLine("创建新的 Student 类对象,完成初始化。");
}

```

前面已经介绍过,`Student` 类的 `RetistrationNmuber` 属性只提供了 `registrationNumber` 私有变量的 `get` 访问器,而没有提供相应的 `set` 访问器。因此,在 `Student` 类的对象创建后,无法再对 `registrationNumber` 变量的值进行修改。所以,只能在对象创建的时候,通过如上构造函数对其进行赋值。使用 `base` 关键字可以指定,在创建对象的时候调用基类的有参数的构造函数,完成必要的赋值过程。在 `Main()` 方法中,可以如下创建 `SchoolStudent` 对象,代码如下:

```


static void Main(string [] args)
{
    SchoolStudent schStd = new SchoolStudent(2, "王华", new DateTime(1985,
4, 1), 500, 486);
    Console.WriteLine("学生{0}的年龄是{1},成绩百分比是{2}%。",
schStd.Name,schStd.GetAge(),schStd.GetPercentage());
}

```

程序执行后,输出结果是:

创建新的学生对象，完成初始化。
 创建新的 school 学生对象，完成初始化。
 学生王华的年龄是 22，成绩百分比是 95.2%。

从以上结果看出，在创建 SchoolStudent 类的对象时，首先调用了 Student 类的有参数的构造函数，之后又调用了 SchoolStudent 类的有参数的构造函数。通过构造函数的调用，完成对类的成员变量的赋值。

 **注意：**在创建子类对象的时候，都将首先调用基类相应的构造函数，这一点在使用继承机制的时候必须予以重视。

8.1.5 使用 protected 访问类型

在 Student 类中，registrationNumber 成员变量是私有（private）类型的，并且只能通过相应属性的 get 访问器进行值的获取。所以，Student 类的使用者，是无法在类外部对 registrationNumber 的值进行修改的。如果期望在 SchoolStudent 子类中对该变量的值进行修改，则在 C# 中是无法完成这个操作的。但是，也不建议将该变量定义为公共（public）的访问类型，因为这样不仅子类能对该变量进行修改，其他类或方法也能对其进行修改，会给数据带来安全性问题。

C# 的 protected（保护）访问类型可以解决这一问题。类的 protected 访问类型变量可以在本类或者该类的子类中对其进行访问，而其他类或方法是无法对该变量进行访问的。而且，protected 访问类型的变量也无法通过对象的引用直接对其访问，否则编译器会产生错误信息。假设类 Tem 有一个 protected 访问类型的方法，代码如下：

```
class Tem
{
    protected void Test( )
    {
        Console.WriteLine("调用 Test( )方法！");
    }
}
```

如果对 Test() 方法进行如下调用：

```
class Sample
{
    static void Main(string [] args)
    {
        Tem a = new Tem( );
        a.Test( );
    }
}
```

在程序编译的时候，会产生如下错误信息：

'Tem.Test()' 不可访问，因为它受保护级别限制

🔔注意：Visual Studio 的版本不同，抛出的异常信息也可能有细微差别。

但是，如果定义一个继承 Tem 类的 SubClass 类，在该类中就可以对 Test()方法进行调用。代码如下：

```
class SubClass : Tem
{
    public SubClass( )
    {
        Test( );
    }
}
```

在继承了 Tem 类的 SubClass 子类中，直接对 Test()方法进行调用。现在进行如下操作：

```
static void Main(string [] args)
{
    SubClass a = new SubClass( );
}
```

执行后的输出结果是：

调用 Test()方法！

也就是说，在子类中成功地对 Test()进行了调用。但是，如果子类的对象直接对基类的 protected 访问类型的方法进行调用也是不允许的。示例代码如下：

```
static void Main(string [] args)
{
    SubClass a = new SubClass( );
    a.Test( ); //会产生错误信息
}
```

🔔说明：与 protected 访问类型相似的还有 protected internal 访问类型，该访问类型也可以支持子类对基类成员变量或者方法的使用。同时，将成员定义成 protected internal 访问类型后，类所在程序集（如 dll 库等）内的其他类或方法也可以对成员进行访问。

8.1.6 用 sealed 关键字终止继承

如果在定义一个类后，不希望它被任何其他类继承，那么，就可以使用 sealed 关键字对类执行密封操作，这样其他的类就无法对该类进行继承。示例代码如下：

```
sealed class Tem
{
    ...//定义类的成员
}
```

```
class Sample : Tem
{
    ...//定义类的成员
}
```

上述代码在编译的时候将会产生如下错误信息：

```
"Program.Sample": 无法从密封类型"Program.Tem"派生
```


8.2 万类之根——Object 类

在 C#中，所有的类、结构体和接口类型都隐式的继承 Object 类，而且数值类型也可以看成从 Object 类派生而来。Object 类定义在 System 命名空间中，能够给所有 .NET framework 环境中的类提供基本的服务和通用功能。也可以说，Object 类是 C#中所有类的根类。

Object 类的引用可以操作所有其他类的对象。Object 类具有以下方法，如表 8.1 所示。

表 8.1 Object类中的方法

方 法	描 述
Equals(Object)	比较两个对象是否相等。默认的实现只支持对引用的比较，也就是说，如果两个引用指向同一个对象，则返回 true。如果对数值类型进行比较，则可以为进行位比较所有的继承类都应该重写该方法，用于实现对该类对象的比较
Static Equals(Object, Object)	与以上方法功能相同，只是该方法是静态方法
GetHashCode()	返回当前对象的 hash 码
GetType()	返回当前对象运行期间的对象类型
static ReferenceEquals(Object, Object)	如果传入的两个引用指向同一个对象则返回 true。否则，返回 false
ToString()	将当前对象转换成字符串后进行输出。所有的继承类都应该重写该类，用于实现将对象转换成字符串的功能
protected Finalize()	该方法是 protected 访问类型的，需要被所有的类重写，用于释放对象所占的资源。该方法在对象被垃圾回收器收回之前，被 CLR 自动调用
protected MemberwiseClone()	提供对当前类的阴影复制，将当前类的所有成员变量的值进行拷贝

注意：Object 类是 C#的多态性（将在 8.3.1 节进行介绍）中非常关键的一个类。

关于 Object 类的示例如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _8._2
{
    //定义 TestClass 类
    public class TestClass
```



```

{
    //公共访问类型方法, 对 ToString( ) 方法的重写
    public override string ToString( )
    {
        return val.ToString( );
    }
    //TestClass 类的构造函数
    public TestClass(int para)
    {
        val = para;
    }
    //私有成员变量
    private int val;
}
//定义 Sample 类
public class Sample
{
    //定义 Sample 类的静态成员方法
    static void ShowValue(object obj)
    {
        Console.WriteLine("传入的 Object 类对象的值是: {0}",
            obj.ToString( ));
    }
    static void Main( )
    {
        int i = 6;
        Console.WriteLine("整数 i 的值是: {0}" , i.ToString( ));
        //隐式的将整形 i 转换为 Object 类型
        ShowValue(i);
        TestClass test = new TestClass(8);
        Console.WriteLine("对象 test 的值是: {0}" , test.ToString( ));
        ShowValue(test);
    }
}
}

```

程序执行后的输出结果是:

```

整数 i 的值是: 6
传入的 Object 类对象的值是: 6
对象 test 的值是: 8
传入的 Object 类对象的值是: 8

```

需要说明的是, 在 Object 类中对 ToString()方法的声明如下:

```
public virtual string ToString( );
```

这是一个公共的 virtual 类型 (将在 8.3.3 节介绍) 方法, 返回一个字符串, 并且没有传入参数。在继承 Object 类之后, 需要对该方法进行重写 (override)。由于所用的值类型 (如 int、char 等) 都是从 Object 类派生的, 因此可以调用 Object 类的 ToString()方法。

示例中的 TestClass 类继承了 Object 类, 这种继承是隐式的, 不需要明确进行声明。因此, 在 TestClass 内部对 ToString()方法进行了重写, 使其可以返回有意义的值。如果不

对该方法进行重写，那么在使用时将执行默认的方法——返回表示类名的字符串。例如，在 `TestClass` 类中不对 `ToString()` 方法进行重写，而直接调用，将输出如下结果：

```
对象 test 的值是: TestClass
```

8.3 C#类的多态特性

8.3.1 什么是多态

通常，学习面向对象程序设计语言，都要经历3个阶段。第1阶段，学习面向对象语言中结构化的程序控制语言；第2阶段，学会编写类、创建对象，以及实现类的继承；第3阶段，就是学会使用多态进行程序设计。本节中将对多态的概念进行介绍。

微软的 MSDN (Microsoft Developer NetWork) 中对多态的解释是：多态性是指类能提供具有功能不同但名称相同的方法的能力，允许对类的某个方法进行调用，而无需考虑该方法所提供的特定实现。

在对类的多态性进行深入介绍之前，首先要说明基类引用能够指向继承类的对象，并实现对继承类对象的操作。以如下代码作为说明：

```
class A
{
    public void MethodA( )
    {
        .....
    }
}
class B : A
{
    public void MethodB( )
    {
        .....
    }
}
```

之后，在使用基类 `A` 和继承类 `B` 的时候，可以进行如下操作：

```
A a = new B( );
```

这样的定义是合法的，基类 `A` 的引用可以指向继承类 `B` 的对象。这时候，可以将继承类 `B` 的对象作为基类 `A` 的对象使用。所以，可以进行如下调用：

```
a.MethodA( );
```

然而，如果进行下述调用则是错误的。

```
a.MethodB( );
```

也就是说，虽然在开始的时候创建的是继承类 `B` 的对象，但是由于使用基类 `A` 的引用指向该对象，所以在使用的時候只能将其作为基类 `A` 的对象使用，即无法对继承类 `B` 中的

成员变量和方法进行访问以及调用。

8.3.2 在基类与子类中定义同名方法

使用示例代码对这种情况进行说明, 首先定义 Shape 类, 该类具有 Draw() 方法, 用于在屏幕上绘制图形。代码如下:

```
class Shape
{
    public void Draw( )
    {
        Console.WriteLine("绘制图形.....");
    }
}
```

之后, 定义继承 Shape 类的 Circle 子类, 该类同样具有用于在屏幕上进行绘制的 Draw() 方法。

```
class Circle : Shape
{
    public void Draw( )
    {
        Console.WriteLine("绘制圆形.....");
    }
}
```


现在, 已经在 Shape 基类以及 Circle 子类中都定义了 Draw() 方法, 两个方法具有相同的名字和参数。如果在 Main() 方法中进行如下操作:

```
static void Main( )
{
    Circle theCircle = new Circle( );
    theCircle.Draw( );
    Shape theShape = new Circle( );
    theShape.Draw( );
}
```

代码运行后, 将输出如下结果:

```
绘制圆形.....
绘制图形.....
```

由代码输出结果可以知道, 虽然两次都是创建 Circle 类的对象, 但是由于指向他们的引用的类型不相同, 导致两次的输出结果并不相同。前者调用的是 Circle 类的 Draw() 方法, 而后者则是调用基类 Shape 的 Draw() 方法。

 **说明:** 在程序设计的时候可能并不期望这样的结果, 而是想通过基类的引用调用继承类中的同名方法。如果要解决这样的问题, 就需要使用 C# 中提供的 virtual 和 override 关键字。

8.3.3 方法重写——virtual 和 override 关键字的使用

如果想对基类 Shape 的 Draw()方法在子类 Circle 中进行重写,需要先将 Shape 基类中的 Draw()方法定义成 virtual 类型,然后再将子类 Circle 中 Draw()方法定义成 override 类型。如下代码所示。

```
class Shape
{
    //定义 Shape 类的 Draw( )成员方法
    public virtual void Draw( )
    {
        Console.WriteLine("绘制图形.....");
    }
}
//Circle 类继承 Shape 类
class Circle :Shape
{
    //定义 Circle 类的 Draw( )成员方法
    public override void Draw( )
    {
        Console.WriteLine("绘制圆形.....");
    }
}
```

现在,在 Main()方法中进行如下操作:

```
static void Main( )
{
    Shape theShape = new Circle( );
    theShape.Draw( );
}
```

代码执行后的输出结果是:


绘制圆形.....

在 Main()方法中,使用基类 Shape 的引用指向子类 Circle 的对象,并且通过该引用调用 Draw()方法。通过输出结果可以知道,程序调用的是子类 Circle 的 Draw()方法,这正是由于对 Draw()方法进行重写所产生的结果。

对类的定义进行分析,因为 Shape 基类的 Draw()方法被指定为 virtual 类型,所以在对程序进行编译的时候,编译器不会立即对 Shape 基类的引用所调用的 Draw()方法进行指定,而是在程序运行时才确定对象的类型以及调用哪个 Draw()方法,也就是 C#中提供的“动态的、运行时对象绑定”机制。

这种处理过程正是面向对象程序设计中的多态性。在基类和子类中,对具有相同方法名和参数的方法进行不同的实现。当使用基类的引用进行方法调用的时候,编译器会在执行的时候判断基类引用所指向对象的准确类型,之后才决定调用哪个方法(基类或子类的)。在进行多态设计的时候,需要首先声明基类的方法为 virtual 类型。只有对声明 virtual 类型

的方法，才能利用多态性在子类中实现对方法的重写。

 **注意：**如果要重写方法，必须在子类中标明将要进行重写的基类的方法为 `override` 类型。通过将方法声明为 `override` 类型，显式地指出对基类方法的重写，才能使编译器确定方法的调用方式。

上述所举的例子相对比较简单，显式地指定了基类的 `theShape` 引用是指向 `Circle` 子类的。代码如下：

```
Shape theShape = new Circle( );
theShape.Draw( );
```

也许有人会觉得，既然已经进行如上指定，那么就没有必要让编译器再进行动态的绑定过程。但是，在实际情况下，程序的设计是十分复杂的，编译器只有在执行的时候才能确定应该调用的对象的方法。下面对上例进行修改，添加新的 `Shape` 类的子类，并在子类中实现不同的 `Draw()` 方法，用于绘制不同的图形。示例代码如下：

```
//Rectangle 类继承 Shape 类
class Rectangle :Shape
{
    //定义 Rectangle 类的 Draw( ) 成员方法
    public override void Draw( )
    {
        Console.WriteLine("绘制矩形.....");
    }
}
//Curve 类继承 Shape 类
class Curve :Shape
{
    //定义 Rectangle 类的 Draw( ) 成员方法
    public override void Draw( )
    {
        Console.WriteLine("绘制曲线.....");
    }
}
```


并且修改 `Main()` 方法中的调用过程：

```
static void Main( )
{
    Shape [] shapes = {new Circle( ), new Rectangle( ), new Curve( )};
    Random random = new Random( );
    //随机调用不同类的 Draw( ) 方法
    for(int i = 0; i < 5; i++)
    {
        int randNum = random.Next(0, 2);
        shapes[randNum].Draw( );
    }
}
```

在程序中，创建了 `Shape` 基类引用的数组，并且数组中的元素分别是 `Circle`、`Rectangle`

和 Curve 子类的对象。之后，通过创建 Random 类的对象用来产生 0 和 2 之间（包括 0 和 2）的整数随机数。产生的随机数被当作 Shape 类型数组的索引，用来进行 Draw()方法的调用。因此，无论是程序设计人员还是编译器，都不可能在程序执行之前知道哪个子类的 Draw()方法将被调用，这就需要编译器具有运行时绑定机制的支持。上述代码执行后将输出如下结果：

```
绘制矩形.....
绘制曲线.....
绘制曲线.....
绘制圆形.....
绘制矩形.....
```

 **注意：**Shape 基类的引用指向何种类型的对象完全是随机的，只有在运行时绑定才能支持程序的正确运行。

8.3.4 用 new 关键字指出重写方法

假设在 Circle 子类中，需要具有 Circle 类自己的 Draw()方法，但是又不希望对基类的 Draw()方法进行重写。在 Java 以及 C++中，都没有办法解决这样的问题。但是，在 C#中可以使用 new 关键字指出某个方法是否重写的方法，避免多态性带来的困扰。

作为示例，如下定义 Shape 类和 Circle 类：

```
class Shape
{
    public virtual void Draw( )
    {
        Console.WriteLine("绘制图形.....");
    }
}
class Circle :Shape
{
    public new void Draw( )
    {
        Console.WriteLine("绘制圆形.....");
    }
}
class Sample
{
    static void Main( )
    {
        Shape theShape = new Circle( );
        theShape.Draw( );
    }
}
```


程序执行后将输出如下结果：

```
绘制图形.....
```


可见,通过在 Circle 类中声明 Draw()方法为 new 类型,避免了 Circle 类中 Draw()方法的多态性。采用这种方式的好处是,如果 Shape 类型具有许多子类,有的子类要求对 Draw()方法进行重写,而另一些子类则不想重写 Draw()方法,使用 new 关键字就可以实现在某些子类中不对 Draw()方法进行重写。

如果在 Shape 基类中声明 Draw()方法为 virtual 类型,而在 Circle 子类中不使用 new 或 override 关键字对方法进行声明,则在编译的时候会产生如下警告信息:

```
"Circle.Draw()"将隐藏继承的成员"Shape.Draw()"
```

说明:如果在基类中声明某个方法为 virtual 类型,那么需要在子类中使用 override 声明对方法的重写,或者使用 new 声明定义自己的同名方法,最好不要不进行任何声明,这点需要在程序设计中予以重视。

8.4 对象类型转换——Up-casting 和 Down-casting

对象的类型转换,可以理解成为“让对象的行为像其他类”或者“改变对象类型表现”。在 C#中,可以将具有继承关系的子类转换为基类(up-casting)或者将基类转换为子类(down-casting)。

8.4.1 子类到基数的 Up-casting 类型转换

Up-casting 转换是比较简单而且安全的,通常可以隐式地执行。事实上,之前的程序中已经使用了这种对象类型的转换。如将基类的引用指向子类的对象,即将子类转换为基类的操作,就是一种 Up-casting 类型转换。代码如下:

```
ParentClass theParent = new ChildClass();
```

8.4.2 基数到子类的 Down-casting 类型转换

与 Up-casting 转换相反,Down-casting 转换是不安全的,而且必须显式指定。所谓不安全,就是指可能会在运行的时候产生异常,从而导致程序运行失败。例如,在之前的程序中定义了如下的 Shape 类型数组:

```
Shape [] shapes = {new Circle(), new Rectangle(), new Curve()};
```

其中, Circle、Rectangle 和 Curve 类都是 Shape 类的子类。如果要使用一个 Rectangle 类型的引用指向 Shape 类型数组中的 Rectangle 类的对象,则不能进行如下的直接赋值:

```
Rectangle rect = shapes[1];
```

而必须显式地进行类型的转换:

```
Rectangle rect =(Rectangle) shapes[1];
```

这种从基类转换为子类的操作，就是 Down-casting 类型转换。需要注意的是，在进行 Down-casting 转换的过程中，可能会导致程序的错误，如下所示。

```
Rectangle rect =(Rectangle) shapes[2];
```

当进行上述转换的时候，试图将 Curve 类型转换为 Rectangle 类型，CLR 在运行的时候将产生如下异常：

```
System.InvalidCastException: Specified cast is not valid.
```

8.4.3 用 is 和 as 关键字做对象类型检查

在 C# 中，可以使用 is 和 as 关键字进行运行时的对象类型检查。is 关键字可以对对象的类型和给定的类型进行比较，如果可以进行类型转换则返回 true，否则返回 false。例如：

```
Console.WriteLine(shapes[1] is Rectangle);
```

代码运行之后将在屏幕上输出 true。而如下代码则会输出 false：

```
Console.WriteLine(shapes[2] is Rectangle);
```

在程序设计的时候，可以使用 is 关键字的这种特性在进行对象类型转换之前检查类型转换是否合法，代码如下：

```
Shape [] shapes = {new Circle( ), new Rectangle( ), new Curve( )};
Rectangle rect = null;
//对对象的类型进行判断
if(shapes[1] is Rectangle)
{
    rect = (Rectangle) shapes[1];
}
```

同样地，也可以使用 as 关键字完成对象类型转换时的检查。不过与 is 关键字不同的是，as 关键字在无法进行类型转换的时候将返回空值（null），而在可以进行类型转换的时候，将直接返回转换后的对象。示例代码如下：

```
Shape [] shapes = {new Circle( ), new Rectangle( ), new Curve( )};
Rectangle rect = shapes[1] as Rectangle;
//对对象的类型进行判断
if(rect != null)
{
    Console.WriteLine("类型转换成功！");
}
else
{
    Console.WriteLine("类型转换不成功！");
}
```


8.5 装箱和拆箱

装箱 (boxing) 和拆箱 (un-boxing) 是使数值类型 (如整数) 能够被当成对象引用的处理过程。数值类型被“装箱”到一个 Object 对象里, 然后通过“拆箱”返回一个数值类型。正是这种处理, 可以对整数调用 ToString() 方法。

装箱可以将一个数值类型的变量隐式地当作一个对象对待。例如, 在声明一个整型变量 i 后, 可以使其调用 ToString() 方法。示例代码如下:

```
int i = 6;  
i.ToString();
```

在进行上述操作的时候, 编译器将自动创建一个 Object 类的对象, 并将 int 类型变量 i 的值 (6) 赋值并装入创建的对象中。之后调用 ToString() 方法, 就是通过 Object 类的对象进行的。上面的隐式的装箱过程也可以写成如下显式的装箱过程:


```
int i = 6;  
Object obj = i; //显式的装箱  
obj.ToString();
```

在进行显式装箱的时候, 不需要进行类型转换, 如下的操作是不必要的:

```
Object obj = (Object) i;
```

另一方法, 拆箱是显式地将一个对象引用转换成为数据类型的变量。装箱可以隐式地进行, 而拆箱则必须显式地进行。下述代码说明了在 C# 中如何进行拆箱操作:

```
int i = 6;  
Object obj = i; //隐式的装箱  
int j = (int) obj; //显式的拆箱
```

 **说明:** 与 Down-casting 类型转换相似, 拆箱也是一种不安全的操作, 如果使用不当可能会在执行的时候产生 InvalidCastException 异常, 读者在使用的时候需要注意这点。

8.6 本章总结

在面向对象的思想中, 继承是一个相当重要的概念。前文已经提到过, 继承是面向对象程序设计的三个基本原则之一, 可见其重要的作用。在 C# 中, 为用户提供了完整的设计良好的继承机制。其中有几点比较关键的内容, 分别是:

- (1) 派生类 (继承类) 对基类的继承。
- (2) 类的成员变量和方法的继承。
- (3) 类的属性及其访问器的继承。

为了给继承增加更大的灵活性,方便程序设计人员完成类的层次结构体系,在C#中定义了密封类的概念,密封类不允许再被继承,密封的方法和属性也不允许被重载。此外,在继承机制中还涉及抽象类的概念,这部分内容将在第9章中进行介绍。抽象和密封的概念是继承机制中的难点,需要理解和掌握。

本章还介绍了C#中多态的概念,“多态性”本身是源于生物学的概念,表示同一种族的生物所具有的不同的特性。将这一概念用到面向对象程序设计中,则表示了同一操作对于不同的类的对象,将表现为不同的处理过程,最后产生不同的处理结果。多态性同样是面向对象程序设计的基本原则之一,在使用C#语言进行程序设计的时候,需要灵活的对继承和多态两个原则进行运用,才能够设计出功能完善、结构严谨的应用程序。

在本章中还介绍了C#中类型转换的概念和其实现,利用类型转换可以增强程序的灵活性。但是,如果不正确的使用类型转换可能会导致在转换的过程中丢失数据,或者引起异常的产生。所以,读者在使用这部分知识的时候需要遵循相应的规则,以免在实际使用的时候发生不必要的错误。

8.7 实战练习

1. 在 Visual Studio 2010 中创建一个控制台应用程序,定义商品类及其多层的派生类。以商品类为基类。第一层派生出服装类、家电类、车辆类。第二层派生出衬衣类、外衣类、帽子类、鞋子类;空调类、电视类、音响类;自行车类、轿车类、摩托车类。要求给出基本属性和派生过程中增加的属性。

2. 在 Visual Studio 2010 中创建一个控制台应用程序,以点(Point)类为基类,重新定义矩形类和圆类。点为直角坐标点,矩形水平放置,由左下方的顶点和长宽定义。圆由圆心和半径定义。派生类操作判断任一坐标点是在图形内,还是在图形的边缘上,或者是在图形外。

第 9 章 抽象类和接口

本章将对抽象类和接口的概念进行介绍。首先，将在本章的前面部分介绍 C# 中抽象方法、抽象类和接口的原理，以及如何实现。然后，将会介绍如何使用 `is` 和 `as` 操作符对接口引用进行转换。


9.1 定义抽象类

抽象类可以简单地理解成为没有完成定义的类。之所以这样说，是因为在抽象类中有一个或多个没有完成过程定义的抽象方法。在抽象类中，只提供对这些抽象方法的方法标识或者方法声明（声明包括方法的方法名和具体参数），而这些方法的实现则在抽象类的继承类中进行。

抽象类和抽象方法可以使用 `abstract` 关键字进行标识，下述代码给出了抽象类和抽象方法的定义示例，代码如下：

```
//定义 Student 抽象类
public abstract class Student
{
    //抽象方法，不含程序主体
    public abstract void GetStudentID( );
    .....//类的其他成员定义
}
```

对于抽象类，由于它是没有完成定义的类，所以无法对抽象类进行实例化操作。也就是说，实例化一个抽象类是非法的。所以，抽象类只有被继承后，才能实现类的相关功能。因此，抽象类是不能使用 `sealed` 关键字对其执行密封操作的，因为 `abstract` 和 `sealed` 在概念上是完全相反的。

 **注意：**如果抽象类被继承后，在子类中必须对抽象类中的所有抽象方法进行实现，否则必须把子类也声明为抽象类。

继承抽象类并对抽象方法进行实现的类，被称做抽象类的具体类（或实体类）。在使用抽象类的时候，可以声明一个抽象类的引用，并将该引用指向一个抽象类的子类对象。下面的代码定义了一个具有两个属性和一个抽象方法的抽象类，代码如下：

```
//定义 TaxCalculator 抽象类
public abstract class TaxCalculator
{
    //定义 TaxCalculator 类的成员变量
```

```

protected double itemPrice;
protected double tax;
//定义TaxCalculator类的抽象成员方法
public abstract double CalculateTax();
//定义TaxCalculator类的Tax属性
public double Tax
{
    get
    {
        return tax;
    }
}
//定义TaxCalculator类的ItemPrice属性
public double ItemPrice
{
    get
    {
        return itemPrice;
    }
}
}

```

在 TaxCalculator 类中定义了两个成员变量，分别是代表商品价格的 itemPrice 和应交税金的 tax。另外，还有一个用来根据商品价格计算 tax 的 CalculateTax()抽象方法。这里把 CalculateTax()方法定义成抽象的，是为了让不同的子类能够根据其商品价格独立计算税金。如果在程序中直接创建 TaxCalculator 类的实例，代码设置如下：

```

static void Main( )
{
    TaxCalculator taxCalc = new TaxCalculator( );
}

```

在编译的时候，将会产生如下错误信息：

无法创建抽象类或接口"TaxCalculator"的实例

为了能够对 TaxCalculator 抽象类进行实例化操作，需要先创建该类的继承类。现在设计如下的 SalesTaxCalculator 类继承 TaxCalculator 抽象类，代码如下：

```

//定义继承 TaxCalculator 类的 SalesTaxCalculator 类
public class SalesTaxCalculator : TaxCalculator
{
    //SalesTaxCalculator 类的构造函数
    public SalesTaxCalculator(double itemPrice)
    {
        this.itemPrice = itemPrice;
    }
    //对 CalculateTax( )抽象方法进行重写
    public override double CalculateTax()
    {
        tax = 0.3 * itemPrice;
        return tax + itemPrice;
    }
}

```



```

    }
}

```

SalesTaxCalculator 类继承了 TaxCalculator 抽象类，并且重写了 CalculateTax()方法。重写的 CalculateTax()方法将商品的税率定为 0.3，并且在计算税金之后返回新的商品价格。同时，SalesTaxCalculator 类具有一个以 itemPrice 为参数的构造函数。如果在 SalesTaxCalculator 类中不对 CalculateTax()方法进行重写，那么在编译的时候会产生如下的错误信息：

```

'SalesTaxCalculator' doesn't implement inherited abstract member
'TaxCalculator.CalculateTax()'

```

所以，必须在 SalesTaxCalculator 类中对 CalculateTax()抽象方法进行重写。在对 SalesTaxCalculator 类进行如上定义之后，就可以对其进行实例化操作。示例代码如下：

```

//定义 Program 类
class Program
{
    static void Main(string[] args)
    {
        SalesTaxCalculator salesTaxCalc = new SalesTaxCalculator(250);
        //调用 CalculateTax( )抽象方法
        double newPrice = salesTaxCalc.CalculateTax( );
        //输出结果
        Console.WriteLine("由于增加税金的原因，商品价格从{0}¥涨到{1}¥！ ",
            salesTaxCalc.ItemPrice, newPrice);
        Console.WriteLine("商品的税金是：{0}¥！ ", salesTaxCalc.Tax);
    }
}

```

程序执行后，将输出如下结果：

```

由于增加税金的原因，商品价格从 250¥涨到 325¥！
商品的税金是：75¥！

```

通过对程序的分析可以知道，当在 SalesTaxCalculator 类中对 CalculateTax()方法重写后，对它的调用与对类的其他方法调用完全一样。

在使用抽象类的时候，可以使用抽象类的引用指向其继承类的对象，并对对象的方法进行调用，代码如下：

```

//定义 Sample 类
public class Sample
{
    static void Main( )
    {
        TaxCalculator taxCalc = new SalesTaxCalculator(250);
        //调用 CalculateTax( )抽象方法
        double newPrice = taxCalc.CalculateTax( );
        //输出结果
        Console.WriteLine("由于增加税金的原因，商品价格从{0}¥涨到{1}¥！ ",
            taxCalc.ItemPrice, newPrice);
        Console.WriteLine("商品的税金是：{0}¥！ ", taxCalc.Tax);
    }
}

```

```

    }
}

```

说明：这种引用方式也是正确的，并且程序在执行后的输出结果与之前的程序输出结果一样。

可以有多个类继承 `TaxCalculator` 抽象类，只要在这些类中提供对 `CalculateTax()` 抽象方法的具体实现即可，代码如下：

```

//定义继承 TaxCalculator 类的 SurchargeTaxCalculator 类
public class SurchargeTaxCalculator : TaxCalculator
{
    //定义 SurchargeTaxCalculator 类的构造函数
    public SurchargeTaxCalculator(double itemPrice)
    {
        this.itemPrice = itemPrice;
    }
    //对 CalculateTax( ) 抽象方法进行重写
    public override double CalculateTax( )
    {
        tax = 0.5 * itemPrice;
        return tax + itemPrice;
    }
}

```

`SurchargeTaxCalculator` 类也可以进行实例化操作，并且可以与 `SalesTaxCalculator` 类共同使用，它们只是对 `CalculateTax()` 方法提供不同的税率。抽象类的使用，可以提高程序设计的灵活性。但是，也要谨慎使用抽象类，避免人为的设计错误。

9.2 定义统一规划的接口


9.2.1 定义接口

接口（interface）是 C# 中一种比较特殊的类型，它用于定义需要在子类中遵守的规范（例如方法的标识）。

接口可以通过 `interface` 关键字进行声明，而且接口同抽象类一样不能进行实例化操作。在接口中可以定义方法（method）、属性（property）或者索引器（indexer）的标识，并且接口中所有的成员都具有 `public` 和 `abstract` 的默认属性。也就是说，接口中所有的方法都必须在子类中进行实现。

接口可以通过类来实现。一个类在实现接口的时候，必须提供对接口中所有成员的定义。并且，在实现接口的时候，使用与继承相同的冒号标识符“：”。通过使用这个标识符，可以表示类对接口的实现。

与类的继承不同的是，一个类可以同时实现多个接口。而且，一个接口可以继承其他接口。通过声明接口类型的引用，可以使该引用指向任意的实现该接口的类。

 **技巧：**在 C# 中定义接口的时候，习惯性的要把接口名的第一个字母设成大写 “I”，作为接口名的标识。例如，IDisposable、ISerializable、ICloneable 和 IEnumerator 等。

下面以 IWindow 接口为例说明如何定义接口，示例如下：

```
//定义 IWindow 接口
interface IWindow
{
    //定义 IWindow 接口的 Position 属性
    Point Position
    {
        get;
        set;
    }
    //定义 IWindow 接口的 Title 属性
    String Title
    {
        get;
        set;
    }
    //定义 IWindow 接口的成员方法
    void Draw();
    void Minimize();
}
```

其中，Position 属性的类型是 Point，该类型的定义如下：

```
//Point 数据类型的定义
public struct Point
{
    public int x;
    public int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```


IWindow 是一个新定义的接口，该接口包含两个具有 get 和 set 访问器的属性，分别是窗体的 Position 属性和 Title 属性。同时，接口中还有可以对窗体进行操作的 Draw()方法和 Minimize()方法的标识。注意，在 IWindow 接口定义的时候，每个成员后面都有一个分号，表明这些成员定义并没有完成。接下来，所有实现该接口的类，都需要实现这些成员。

在使用接口的时候，必须明白接口声明了抽象的成员标识，实现接口的类必须实现这些标识。否则，那些继承接口，却没有实现接口的成员类是无法通过编译的。现在，定义 RectangularWindow 类实现 IWindow 接口，示例代码如下：

```
//继承 IWindow 接口的 RectangularWindow 类
```

```
public class RectangularWindow : IWindow
{
    //私有成员变量
    string title;
    Point position;
    //RectangularWindow 类的构造函数
    public RectangularWindow(string title, Point position)
    {
        this.title = title;
        this.position = position;
    }
    //实现 IWindow 接口中的 Position 属性
    public Point Position
    {
        get
        {
            return position;
        }
        set
        {
            position = value;
        }
    }
    //实现 IWindow 接口中的 Title 属性
    public string Title
    {
        get
        {
            return title;
        }
        set
        {
            title = value;
        }
    }
    //实现 IWindow 接口中的成员方法
    public void Draw( )
    {
        Console.WriteLine("绘制矩形窗体!");
    }
    public void Minimize( )
    {
        Console.WriteLine("最小化矩形窗体!");
    }
}
```

如上所示, 可以使用冒号操作符表示 RectangularWindow 类对 IWindow 接口的实现。由于 RectangularWindow 类是对 IWindow 接口的实现, 并且没有声明 RectangularWindow 类是抽象类, 所以必须在该类中实现所有的接口成员。这些成员包括 Title 和 Position 属性, 以及 Draw() 和 Minimize() 方法。

 **注意：**在 `RectangularWindow` 类中实现这些成员的时候，把它们都声明为 `public` 类型。而在接口中声明的时候，却没有把这些成员声明为 `public` 类型。

如果在接口成员实现的时候，不将其声明为 `public` 类型，代码如下：

```
void Draw( )
{
    Console.WriteLine("绘制矩形窗体!");
}
```

那么在编译的时候，将会产生如下错误信息：

```
"RectangularWindow" 不 实 现 接 口 成 员 "IWindow.Draw()" 。
"RectangularWindow.Draw()" 无法实现接口成员，因为它不是公共的。
```

编译器支持对接口实现的检查，并能够返回可能的错误原因。事实上，前面已经提到过，接口内的所有成员都默认是 `abstract` 和 `public` 类型的。所以，在对接口进行多态实现的时候，不能降低成员原有的访问类型。正是这个原因，在实现接口的时候，必须将成员的访问类型声明为 `public` 类型。

另一方法中需要注意的是，在重写接口的抽象方法时，并没有使用 `override` 关键字。虽然是在使用 C# 的多态性，但是如果在实现方法的时候在前面添加 `override` 声明，程序编译的时候将产生如下错误信息：

```
"RectangularWindow.Draw()": 没有找到适合的方法来重写
```

这里不使用 `override` 关键字，是因为这是对接口成员的实现，而不是对其的重写。读者需要在实际使用的时候注意。

9.2.2 一个类实现多个接口

一个类可以实现一个或多个接口。在这种情况下，这个类必须提供对这些接口中所有成员的实现。假设有如下两个接口：


```
//定义 ILoggable 接口
interface ILoggable
{
    //ILoggable 接口中的 Log( ) 方法
    void Log(string filename);
}
//定义 IFile 接口
interface IFile
{
    //IFile 接口的成员方法
    string ReadLine();
    void Write(string s);
}
```

`ILoggable` 接口中只有一个 `Log()` 方法，用于在文件中记录活动信息。而 `IFile` 接口有两

个方法，分别是用于读取文件的 `ReadLine()` 方法和用于向文件写入内容的 `Write()` 方法。下面定义 `MyFile` 类对这两个接口进行实现，示例代码如下：

```
//定义继承 ILoggable 接口和 IFile 接口的 MyFile 类
public class MyFile : ILoggable, IFile
{
    //私有成员变量
    private string filename;
    //MyFile 类的构造函数
    public MyFile(string filename)
    {
        this.filename = filename;
    }
    //实现 ILoggable 接口的 Log( ) 方法
    public void Log(string filename)
    {
        Console.WriteLine("在 '{0}' 文件中记录活动信息!", filename);
    }
    //实现 IFile 接口的 ReadLine( ) 方法
    public string ReadLine( )
    {
        return "读出 MyFile 中的一行! ";
    }
    //实现 IFile 接口的 Write( ) 方法
    public void Write(string s)
    {
        Console.WriteLine("向 '{0}' 文件中写入: {1}", filename, s);
    }
}
```

`MyFile` 类同时实现了 `ILoggable` 和 `IFile` 两个接口。

 **说明：**在同时实现多个接口的时候，可以使用逗号分隔符“，”将要实现的多个接口分隔开来。

在 `MyFile` 类中实现了 `ILoggable` 和 `IFile` 两个接口中的 3 个方法，并且还具有一个以文件名字符串为参数的构造方法。可以像正常类一样对 `MyFile` 类进行操作，示例代码如下：

```
//定义 Sample 类
public class Sample
{
    static void Main( )
    {
        //创建 MyFile 类的对象
        MyFile aFile = new MyFile("myfile.txt");
        //调用 aFile 对象的成员方法
        aFile.Log("C:\\csharp.txt");
        aFile.Write("Hello World!");
        Console.WriteLine(aFile.ReadLine( ));
    }
}
```


程序执行后的输出结果是：

```
在'C:\csharp.txt'文件中记录活动信息！
向'myfile.txt'文件中写入：Hello World！
读出 MyFile 中的一行！
```

从程序的输出结果可以看出，MyFile 类成功实现了 ILoggable 接口和 IFile 接口。但是，如果在 ILoggable 接口中将 Log() 方法的名字改成 Write() 方法，结果会怎样呢？示例代码如下：

```
interface ILoggable
{
    void Write(string filename);
}
interface IFile
{
    string ReadLine();
    void Write(string s);
}
```

如果 MyFile 类同时实现 ILoggable 和 IFile 两个接口，这两个接口中都有 Write() 方法，并且两个方法具有相似的方法标识，这就需要指定具体的接口名称来对同名方法进行实现。

9.2.3 实现多接口同名方法

如果一个类同时对多个接口实现，而且在这些接口中有两个以上的接口存在相同标识的方法，那么就需要在实现方法的时候指定是在实现哪个接口的方法。这种指定可以通过在方法名前增加接口名前缀来实现，在接口名和方法名之间可以使用“.”操作符。

如 9.2.2 节中定义的 ILoggable 和 IFile 接口，两个接口中都有相同标识的 Write() 方法，可以使用 MyFile 类对两个接口进行如下实现：

```
//定义继承 ILoggable 接口和 IFile 接口的 MyFile 类
public class MyFile : ILoggable, IFile
{
    //私有成员变量
    private string filename;
    //MyFile 类的构造方法
    public MyFile(string filename)
    {
        this.filename = filename;
    }
    //实现 ILoggable 接口的 Write( ) 方法
    public void ILoggable.Write(string filename)
    {
        Console.WriteLine("在'{0}'文件中记录活动信息！", filename);
    }
    //实现 IFile 接口的 ReadLine( ) 方法
    public string ReadLine( )
    {
        return "读出 MyFile 中的一行！";
    }
}
```

```

    }
    //实现 IFile 接口的 Write( ) 方法
    * public void Write(string s)
    {
        Console.WriteLine("向'{0}'文件中写入: {1}", filename, s);
    }
}

```

通过上例可以看出，在程序中显式地指定了对 `ILoggable` 接口的 `Write()` 方法的实现。通过使用接口名进行指定之后，编译器就可以在类中区分对不同的 `Write()` 方法的定义。但是如果在 `Main()` 方法中进行如下调用：

```

static void Main( )
{
    MyFile aFile = new MyFile("myfile.txt");
    //调用 aFile 对象的成员方法
    aFile.Log("C:\\csharp.txt");
    aFile.Write("Hello World!");
    Console.WriteLine(aFile.ReadLine());
}

```

程序执行后的输出结果将是：

```

向'myfile.txt'文件中写入: C:\\csharp.txt
向'myfile.txt'文件中写入: Hello World!
读出 MyFile 中的一行!

```

很明显，程序的输出结果并不是之前所期望的。程序两次都调用了 `IFile` 接口的 `Write()` 方法，而没有调用 `ILoggable` 接口的 `Write()` 方法，这主要是在 `Main()` 方法中对 `Write()` 方法的调用方式不正确造成的。在调用 `ILoggable` 接口的 `Write()` 方法的时候，必须使用 `ILoggable` 接口的引用指向 `MyFile` 类的对象，形成类型的转换，代码如下：

```

ILoggable logger = aFile;
log.Write("C:\\csharp.txt");

```

由于可以将 `ILoggable` 接口看成 `MyFile` 类的父类，所以可以使用 `ILoggable` 接口的引用指向 `MyFile` 类的对象，从而实现调用 `ILoggable` 接口中 `Write()` 方法的目的。正确的 `Main()` 方法中的调用应如下所示：

```

static void Main( )
{
    MyFile aFile = new MyFile("myfile.txt");
    aFile.Write("Hello World!");
    //使用 ILoggable 接口类型的引用指向 aFile 对象
    ILoggable logger = aFile;
    log.Write("C:\\csharp.txt");
    Console.WriteLine(aFile.ReadLine( ));
}

```

程序执行后，将输出如下结果：

```

向'myfile.txt'文件中写入: Hello World!

```


在'C:\\csharp.txt'文件中记录活动信息!
 读出 MyFile 中的一行!

9.2.4 使用 is 和 as 操作符实现接口转换

在上例中，将 MyFile 类的对象 aFile 转换成了 ILoggable 接口类型。通过前面介绍的知识可以了解到，这种转换是继承类中经常使用的 up-casting 类型转换。由于在实际工作中，这种类型转换会经常被使用，所以需要讨论这种类型转换的可行性。使用 is 和 as 操作符的目的，就是判断在运行期间是否可以进行这种接口对象类型的转换（其实，相关内容在继承和多态一章已经讲解过）。


is 操作符可以用来检查对象的类型，并返回 boolean 类型的判断结果。如果对象的类型与指定的类型相符，则返回 true，否则将返回 false，例如：

```
MyFile aFile = new MyFile("myfile.txt");
Console.WriteLine(aFile is ILoggable);
```

由于 MyFile 类是 ILoggable 接口的子类型，所以上述代码在执行后将打印“true”。如果对象与指定的类型不一致，则会返回相反的结果，如下所示。

```
string s = "Hello World!";
Console.WriteLine(s is ILoggable);
```

代码执行后，会打印“false”的信息。与 is 操作符相似，as 操作符也可以对对象进行指定类型的判断。不过，使用 as 操作符进行判断之后，如果对象类型与指定类型相符，则会返回对象转换类型后的引用。如果不相符，则会返回空值（null），示例代码如下：

 **注意：**is 操作符与 as 操作符作用相似，返回值不同。

```
MyFile aFile = new MyFile("myfile.txt");
ILoggable logger = aFile as ILoggable;
//判断是 logger 类型的转换是否成功
if(logger == null)
{
    Console.WriteLine("类型转换不成功!");
}
else
{
    Console.WriteLine("类型转换成功!");
}
```

上述程序在执行后，将会打印如下结果：

类型转换成功!


也就是说，aFile 是 ILoggable 接口的子类型的对象。进行 as 操作后，将返回一个指向 aFile 对象的 ILoggable 接口类型的引用。而如果将代码进行如下修改：

```
Sample s = new Sample();
ILoggable logger = s as ILoggable;
```

```
//判断 logger 类型的转换是否成功
if(logger == null) //判断 logger 类型的转换是否成功
{
    Console.WriteLine("类型转换不成功!");
}
else
{
    Console.WriteLine("类型转换成功!");
}
```

程序执行后，打印的结果将变成：

类型转换不成功！

 **技巧：**如果需要在程序设计的时候使接口类型的引用指向相关的对象，最好进行类型能否转换的检查，以保证程序执行的正确性。

9.2.5 接口间能继承吗

通常情况下，一个接口也可能是继承其他一个或多个接口得到的。假如之前提到的 IFile 接口存在对 IReadable 和 IWritable 接口的继承，那么可以进行如下定义：


```
//定义 IWritable 接口
interface IWritable
{
    //定义 IWritable 接口的成员方法
    void Write(string s);
}
//定义 IReadable 接口
interface IReadable
{
    //定义 IReadable 接口的成员方法
    string ReadLine( );
}
//定义继承 IWritable 接口和 IReadable 接口的 IFile 接口
interface IFile :IWritable, IReadable
{
    //定义 IFile 接口的成员方法
    void Open(string filename);
    void Close( );
}
```

在上述程序中，分别定义了 IWritable 接口、IReadable 接口和 IFile 接口。在 IWritable 接口中定义了 Write(string)方法，该方法可以用来向文件中写入指定的字符串。而在 IReadable 接口中定义了 ReadLine()方法，调用 ReadLine()方法后可以从文件中返回一个字符串。IFile 接口继承了 IWritable 接口和 IReadable 接口，而且还定义了两个新的方法。这两个方法分别是 Open(string)方法和 Close()方法，可以用来打开或关闭某个文件。

在定义完如上接口之后，创建 MyFile 类实现 IFile 接口。在 MyFile 类中提供了对 IFile

接口中方法的实现，其中包括 IFile 接口本身定义的方法和继承其他接口的方法，示例代码如下：

```
//定义继承 IFile 接口的 MyFile 类
public class MyFile : IFile
{
    //私有成员变量
    private string filename;
    //实现 IFile 接口的 Open( )方法
    public void Open(string filename)
    {
        this.filename = filename;
        Console.WriteLine("打开'{0}'文件!", filename);
    }
    //实现 IReadable 接口的 ReadLine( )方法
    public string ReadLine( )
    {
        return "读出 MyFile 中的一行!";
    }
    //实现 IWritable 接口的 Write( )方法
    public void Write(string s)
    {
        Console.WriteLine("向'{0}'文件中写入: {1}", filename, s);
    }
    //实现 IFile 接口的 Close( )方法
    public void Close( )
    {
        Console.WriteLine("关闭'{0}'文件!", filename);
    }
}
```

说明：在 MyFile 类中提供了对 IFile 接口内方法的简单实现，即方法通过打印消息的方式表示方法被实现和调用。

下面的 Sample 类中包含了程序的 Main()方法，具体实现过程如下：

```
//定义 Sample 类
class Sample
{
    static void Main( )
    {
        MyFile aFile = new MyFile( );
        //调用 aFile 对象的方法
        aFile.Open("C:\\csharp.txt");
        aFile.Write("Hello World!");
        Console.WriteLine(aFile.ReadLine( ));
        aFile.Close( );
    }
}
```

在 Main()方法中创建了 MyFile 类的实例，并且定义了名为 aFile 的引用。之后，就可

以使用 `aFile` 引用调用对象的成员方法。

程序执行后的输出结果如下：

```
打开'C:\csharp.txt'文件!  
向'C:\csharp.txt'文件中写入: Hello World!  
读出 MyFile 中的一行!  
关闭'C:\csharp.txt'文件!
```

从输出结果中可以分析出方法的调用过程和执行结果。

9.3 本章总结

本章介绍了 C# 中接口的概念, 简单地说, 接口是一组包含了函数原型方法的数据结构。通过这组数据结构, 用户可以提供必要的功能实现。接口可以从父接口中继承, 不过这种继承是说明性继承, 而不是实现性继承。也就是说, 接口只说明了子类中需要实现的方法原形, 方法的实现需要子类来进行。而且, 接口的继承可以是多继承的。

在进行 C# 程序设计时, 可以通过类来实现接口中定义的方法和属性等成员。继承接口的类必须为接口的所有成员提供具体的实现, 包括接口中显示声明的各种成员, 以及接口从父接口中继承的成员。不过, 需要注意的是, 子类可以对基类已经实现的接口进行重新实现。


有时, 基类并不与具体的事物相联系, 而只是为了表达某类抽象的概念, 为其他的类提供一个具有公共属性的基类。抽象类有以下几点需要注意的知识:

- (1) 抽象类只能作为其他类的基类, 不能直接进行实例化操作。
- (2) 抽象类中可以包含抽象成员。
- (3) 抽象类不能同时被声明为密封类。

同时, 抽象类也可以对接口进行实现, 但必须保证接口成员能够影射到抽象类的抽象成员。如果抽象类的继承类是非抽象类, 那么必须对接口的成员进行实现。

9.4 实战练习

1. 在 Visual Studio 2010 中创建一个控制台应用程序, 使用面向对象的思想设计自定义类和接口, 编写代码描述飞机和麻雀。

 **提示:** 首先分析飞机和麻雀的共性是都能飞, 则可以定义飞行的接口, 然后定义飞机、麻雀类分别实现飞行的接口。

2. 在 Visual Studio 2010 中创建一个控制台应用程序, 在程序中创建一个名称为 `Vehicle` 的接口, 在接口中添加两个带有一个参数的方法 `start()` 和 `stop()`。在两个名称分别为 `Bike` 和 `Bus` 的类中实现 `Vehicle` 接口。创建一个名称为 `interfaceDemo` 的类, 在 `interfaceDemo` 的 `main()` 方法中创建 `Bike` 和 `Bus` 对象, 并访问 `start()` 和 `stop()` 方法。

第 10 章 数组与集合

本章将对 C# 中数组、集合，以及字符串的知识进行介绍，讲解它们的概念和使用方法。在章节开始，首先对多维数组进行说明，其中将重点介绍长度固定数组（矩形数组）和长度可变数组。然后，将对 C# 中集合的知识进行介绍，包括如何对其实现和访问。


10.1 数组的数组——多维数组

10.1.1 认识多维数组

在第 6 章中已经介绍过，数组是一系列相同数据类型元素的集合。在 C# 中，数组的元素可以是对象也可以是引用类型，所以数组是在堆上存储的。一维数组的相关知识已经介绍过，本章将主要介绍 C# 中多维数组的概念和使用方法。

多维数组可以看做“数组的数组”。也就是说，多维数组中的一个元素可以是其他的数组。这与数据库中的表十分相似，表中的行元素是列元素的集合。如果一个多维数组的列中，只有一个元素，而不是元素的集合，那么该多维数组被称为二维数组（多维数组中最普通的类型）。否则，如果数组的列元素是其他的数组，那么这样的数组被称为 n 维数组， n 是数组的深度。在 C# 中，多维数组可以分成两种类型：

- (1) 矩形数组（数组的每个行元素都包含相同个数的列元素）；
- (2) 变长数组（数组的每个行元素可以包含不同数量的列元素）。

说明：在数组中，也可以使用行与列的概念对数组中的元素进行划分。

下面的例子给出了不同类型数组的结构示意图，并且标识了数组中元素的索引。需要注意的是，任何类型数组的第 1 个元素的索引值都是 0。

如图 10.1 是一维数组的结构示意图，可以看到一维数组通过一维索引值就可以得到具体元素的数值。一维数组创建和使用都比较简单，是程序设计过程中经常使用的数组类型。

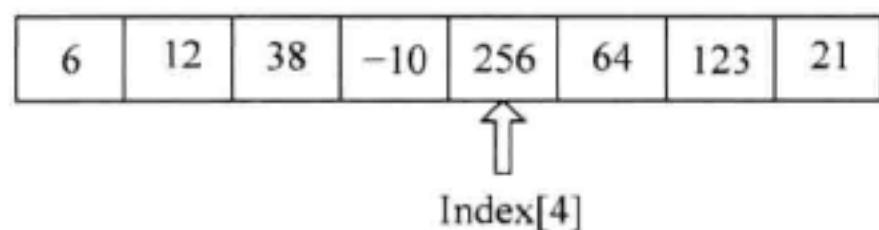


图 10.1 一维数组示意图

如图 10.2 所示，二维数组需要通过两位索引查找数组中的元素。其中第 1 位是数组的行索引，第 2 位是数组的列索引。与一维数组相似，二维数组的行/列索引的起始值都是 0，

最大值是数组的行/列数减1。图 10.2 所示的数组是一个 4×8 的数组，也就是说数组共有 4 行、8 列，所以数组行索引的最大值是 3，而列索引的最大值是 7。可以看到，`Index[1][7]` 就代表数组第 2 行最后一个元素。上例是一个矩形二维数组，该数组中每一行中都有 8 个元素，在计算数组元素位置的时候相对比较简单。

如图 10.3 所示的二维数组是一个变长数组，也就是说，数组每一行的元素数量是不固定的。变长二维数组也是通过两位索引值进行元素定位的，图中 `Index[3][5]` 指向的就是数组的最后一个元素。在使用变长数组的时候，需要注意数组索引的正确使用。

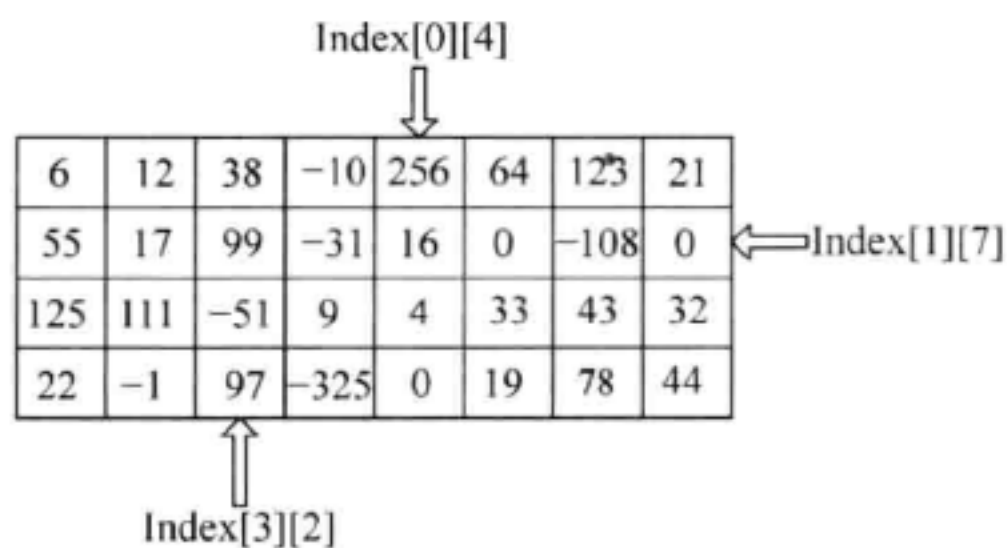


图 10.2 矩形二维数组示意图

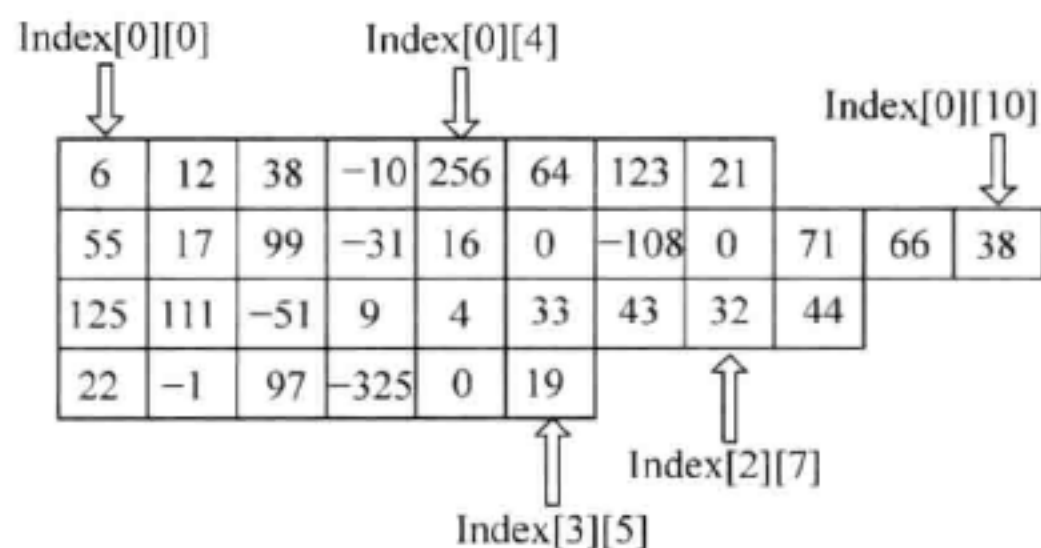


图 10.3 变长二维数组示意图

图 10.4 中的数组是一个 $3 \times 2 \times 3$ 的三维数组，可以通过三位索引值对数组中的元素进行定位，如图中所示。三维数组可以看做二维数组的元素中保存着一个一维数组，上例的三维数组就能看成是一个 3×2 的二维数组的元素，是一个有 3 个元素的一维数组。

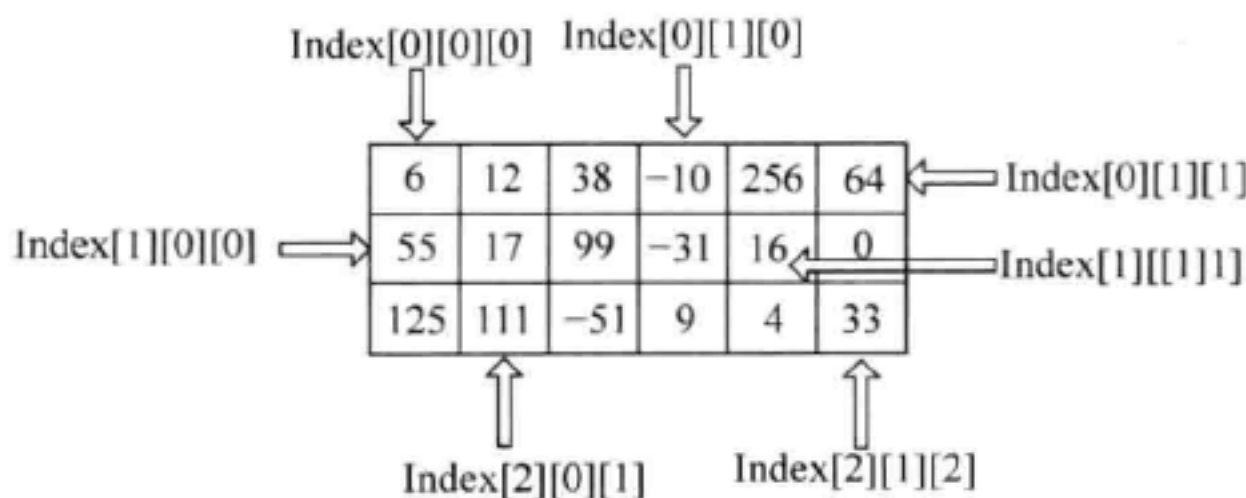


图 10.4 三维数组示意图

说明：其他 n 维数组也可以按照上述方法进行划分。

10.1.2 怎样实例化多维数组

前面介绍过，可以使用下述方式进行一维数组的创建：

```
int [] intArray = new int[6];
```

通过如上操作之后，就可以建立一个具有 6 个元素的 `int` 类型的一维数组。在数组创建之后，可以对数组中的元素进行如下访问：

```
intArray[0] = 6;
intArray[3] = -11;
```



```
intArray[5] = 121;
```

上述操作是对数组元素的赋值。多维数组的创建与一维数组创建的操作非常相似，但在使用的时候需要注意多维数组与一维数组之间的差别。例如，想创建一个 2×3 的矩形二维数组时，可以进行如下操作：

```
int [ , ] myTable = new int[2, 3];
```

在创建数组之后，数组中的元素都进行默认的赋值。因此，在完成创建之后，myTable 数组元素的初值都是 0。可以通过循环访问的方式，输出 myTable 数组元素的值，代码如下：

```
//输出 myTable 数组中的元素
foreach(int intVal in myTable)
{
    Console.WriteLine(intVal);
}
```

上述的代码执行之后，将输出 myTable 数组中元素的值：

```
0
0
0
0
0
0
0
```

在多维数组创建之后，可以对数组元素进行赋值。这就需要使用数组索引对数组元素进行定位，如果想对 myTable 数组的第 1 个元素进行赋值，可以如下进行：

```
myTable[0, 0] = 32;
```

同样地，也可以使用这种方法对数组中其他的元素进行赋值：

```
myTable[0, 1] = 2;
myTable[0, 2] = 21;
myTable[1, 0] = 16;
myTable[1, 1] = -1;
myTable[1, 2] = 6;
```


现在，可以使用二重循环对 myTable 数组中的元素进行输出，代码如下：

```
//输出 myTable 数组中的元素
for(int row = 0; row < myTable.GetLength(0); row++)
{
    for(int col = 0; col < myTable.GetLength(1); col++)
    {
        Console.WriteLine("{0}行{1}列数组元素的值是: {2}", row, col,
            myTable[row, col]);
    }
}
```

在上述代码中，使用了二重循环用于获取 myTable 数组中的所有元素。同时，在程序

中还使用了 `System.Array` 类的 `GetLength()` 方法，用于获取数组指定维数的长度。程序在执行后将输出如下的结果：

```
0 行 0 列数组元素的值是： 32
0 行 1 列数组元素的值是： 2
0 行 2 列数组元素的值是： 21
1 行 0 列数组元素的值是： 16
1 行 1 列数组元素的值是： -1
1 行 2 列数组元素的值是： 6
```

说明：这里使用的 `System.Array` 类是 .NET 中默认的对数组进行操作的类，在创建数组后就可以调用该类中的方法对数组进行操作。

10.1.3 什么是变长数组

变长数组每一行的数组长度是不一样的，所以变长数组在创建的时候与矩形数组存在一定的差别。例如，可以创建一个 3 行的数组，第 1 行的长度是 3，第 2 行的长度是 5，而第 3 行的长度是 2，在创建这样的变长数组的时候可以进行如下的操作：

```
int [][] myTable = new int[3][];
myTable[0] = new int[3];
myTable[1] = new int[5];
myTable[2] = new int[2];
```

在创建变长数组之后，就可以对数组中的元素进行赋值操作，代码如下：

```
myTable[0][0] = 32;
myTable[0][1] = 2;
myTable[0][2] = 21;

myTable[1][0] = 16;
myTable[1][1] = -1;
myTable[1][2] = 6;
myTable[1][3] = 101;
myTable[1][4] = -56;

myTable[2][0] = 12;
myTable[2][1] = 8;
```

同样，可以对变长数组中的所有元素进行循环读取，操作如下：

```
//输出 myTable 数组中的元素
foreach(int [] row in myTable)
{
    foreach(int col in row)
    {
        Console.WriteLine(col);
    }
    Console.WriteLine();
}
```


上述代码的功能比较简单，就是通过两个 `foreach` 循环遍历数组中的所有元素。其中，第一个 `foreach` 循环用于得到数组中的每一行（每一行都是一个 `int` 类型的数组），而第二个 `foreach` 循环则是用于将每一行中的数组元素进行打印输出。程序执行后将输出如下的结果：

```
32;
2;
21;

16;
-1;
6;
101;
-56;

12;
8;
```

前面还介绍了对矩形二维数组以及变长二维数组的创建与操作，其他多维数组也可以进行类似的操作。下面以三维数组为例进行说明，首先是矩形三维数组的相关操作，代码如下：

```
int [ , , ] myTable = new int[3, 2, 4];

myTable[0, 0, 0] = 3;
myTable[1, 1, 1] = 6;
```


可见，对多维矩形数组的操作与二维矩形数组十分相似，只要在使用的时候灵活运用就可以了。接下来举例说明变长三维数组的相关操作，代码如下：

```
int [ ] [ ] [ ] myTable = new int[2] [ ] [ ];
myTable[0] = new int[2] [ ];
myTable[0][0] = new int[3];
myTable[0][1] = new int[4];

myTable[1] = new int[3] [ ];
myTable[1][0] = new int[2];
myTable[1][1] = new int[4];
myTable[1][2] = new int[3];

myTable[0][0][0] = 34;
myTable[0][1][1] = 43;
myTable[1][2][2] = 76;
```

上述的代码中创建了 `myTable` 三维变长数组，可以看到它是由两个二维数组组成的。第一个二维数组是行数为 2 的变长数组，每行的长度分别为 3 和 4。第二个二维数组也是变长数组，其行数是 3，每行的长度分别是 2、4 和 3。完成对数组的创建之后，对数组中的部分元素进行了赋值操作，代码中没有对所有的元素进行赋值，只是举例说明对几个不同位置元素的赋值操作。

 **说明：**在程序设计过程中，多维变长数组比较不容易管理和操作。但是，多维变长数组对解决一些复杂的问题十分有用。所以，在设计程序时难免会使用到这种数组。不过，只要真正理解多维数组就是“数组的数组”，那么在实际使用时将会减少一些不必要的困扰。

在前面的例子中，将多维数组都定义成 `int` 类型的，目的是为了进行说明。而事实上，数组可以被定义成多种类型。例如，可以创建字符串类型的数组，或者是用户定义的类的对象的数组。下述代码就是创建字符串类型的数组：

```
string [] cities = new string[4];
```

其他类型（包括类的对象）的数组也可以这样进行创建。上例中字符串数组在创建的时候，可以采用多种方式对数组进行初始化操作。例如：

```
string []cities = new string[4] {"Beijing", "Shanghai", "Shenzhen",  
"Nanjing"};
```

或者

```
string []cities = new string[] {"Beijing", "Shanghai", "Shenzhen", "Nanjing"};
```

或者

```
string []cities = {"Beijing", "Shanghai", "Shenzhen", "Nanjing"};
```

此外，还可以将数组引用的声明和初始化操作分开进行，示例如下：

```
string []cities;  
cities = new string[4] {"Beijing", "Shanghai", "Shenzhen", "Nanjing"};
```

同样，在创建二维数组的时候，也可以直接对其进行初始化操作。

```
string [][]cityLists = {new string[] {"Beijing", "Shanghai"}, new  
string[] {"Shenzhen", "Nanjing", "Tianjing"}};
```

关于数组，还有以下重要的属性和方法：

- (1) `Length` 属性包含了数组各维中的元素数量。
- (2) `GetLength()` 方法给出了数组某一维中的元素数量。
- (3) `GetUpperBound()` 方法给出了数组某一维的索引上限。
- (4) `GetLowerBound()` 方法给出了数组某一维的索引下限。

10.2 用 foreach 循环进行数组的遍历

在前面的内容中，已经多次使用过 `foreach` 循环遍历数组中的元素。本节中将对 `foreach` 循环的原理进行说明，并介绍如何使用 `foreach` 循环实现对数组的遍历。为了这个目的，需要先设计实现 `IEnumerable` 接口。在 `IEnumerable` 接口中只有一个 `GetEnumerator()` 方法，调用该方法后可以返回 `IEnumerator` 类型的对象。在 `IEnumerator` 接口中包含一个公共访问类型的 `Current` 属性和两个公共成员方法 `MoveNext()` 及 `Reset()`。在 `IEnumerator` 接口中对


Current 属性的定义如下:

```
public object Current
{
    get;
}
```

调用该方法后能够得到数组或集合当前元素的对象类型。接口中 MoveNext()方法的声明如下:

```
bool MoveNext( );
```

调用该方法后可以更改选定的数组或集合的元素,将选定的元素改为当前选定元素之后的数组或集合元素。如果成功地更改选定的数组或集合元素,则方法返回 true 值。否则,如果当前已经选定数组或集合的最后一个元素,调用方法后将返回 false 值。

 **说明:** 在首次调用 MoveNext()方法后,将选定数组或集合的第一个元素。因此,只有在调用 MoveNext()方法后才可以访问 Current 属性,从而获得当前选定的数组或集合的元素。

最后,在 IEnumerator 接口中还将声明 Reset()方法:

```
void Reset( );
```

该方法可以重置 IEnumerator 接口的对象,将其赋为初值。在进行重置之后,调用 MoveNext()方法将重新选定数组或集合的第一个元素。

接下来将举例说明如何实现 IEnumerable 和 IEnumerator 接口,从而能够对类使用 foreach 循环遍历其中的元素,代码如下:

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _10._2
{
    //定义 Sample 类
    class Sample
    {
        static void Main( )
        {
            MyList list = new MyList( );
            //输出 list 中的数据
            foreach(string name in list)
            {
                Console.WriteLine(name);
            }
        }
    }
    //定义继承 IEnumerable 接口的 MyList 类
    class MyList : IEnumerable
```

```

{
    //静态成员变量
    static string []citiesName = {"Beijing","Shanghai",
    "Shenzhen","Nanjing"};
    //定义 GetEnumerator( )方法
    public IEnumerator GetEnumerator( )
    {
        return new MyEnumerator( );
    }
    //定义继承 IEnumerator 接口的 MyEnumerator 私有类
    private class MyEnumerator : IEnumerator
    {
        int index = -1;
        //定义 Current 属性
        public object Current
        {
            get
            {
                return citiesName[index];
            }
        }
        //定义 MyEnumerator 类的成员方法
        public bool MoveNext( )
        {
            //判断 index 的值
            if(++index >= citiesName.Length)
            {
                return false;
            }
            else
            {
                return true;
            }
        }
        public void Reset( )
        {
            index = -1;
        }
    }
}
}


```

程序中定义了 `MyList` 类，并且在该类中内嵌了名为 `MyEnumerator` 的私有成员类。`MyEnumerator` 类是对 `IEnumerator` 接口的实现，在 `MyEnumerator` 类中实现了 `IEnumerator` 接口中公共访问类型的属性和方法。此外，`MyList` 类实现了 `IEnumerable` 接口，其 `GetEnumerator()` 方法的返回值是 `MyEnumerator` 类型的对象。同时，`MyList` 类定义了一个名为 `citiesName` 的字符串数组。在 `MyEnumerator` 类中实现了对该数组的遍历，在遍历的过程中使用 `index` 整形变量记录当前的数组元素。`index` 变量的初值是-1，每当成功调用一次 `MoveNext()` 方法后，`index` 的值都会增1。`MoveNext()` 方法调用是否成功与当前选定数组元素的位置有关，如果当前元素已经是数组的最后一个元素，就会返回 `false`，如果方法调

用成功，将返回 `true` 值。`MyEnumerator` 类的 `Current` 属性可以返回当前选定的数组元素，而 `Reset()` 方法则可以重置 `index` 变量的值为 `-1`，这样就可以从数组首元素再次开始遍历。

在 `Sample` 类中，首先创建了 `MyList` 类的实例，然后使用 `foreach` 循环对 `MyList` 类的对象进行了遍历。上述程序执行后的输出结果是：

```
Beijing
Shanghai
Shenzhen
Nanjing
```

 说明：之所以能够对 `MyList` 类的对象实现遍历，就是因为 `MyList` 类实现了 `IEnumerable`。

10.3 比数组灵活好用的集合

在 C# 中，集合是一个非常重要的概念。虽然使用数组同样可以表示相关对象的集合，但是使用数组代表集合的时候还是会遇到一些限制。首先，数组的大小通常是固定的，而且需要在数组创建的时候就进行限定。其次，数组只能包含相同数据类型的对象，并且同样要在数组创建的时候进行定义。此外，数组没有提供对其元素进行插入和查找的特殊机制。为了解决这些问题，C# 和 .NET Framework Class Library (FCL) 提供了一系列的类用于处理不同类型的集合。

这些集合类都定义在 `System.Collections` 命名空间中，在该命名空间中最常使用的类如表 10.1 所示。

表 10.1 `System.Collections` 命名空间中的集合类

类 名	描 述
<code>ArrayList</code>	与数组相似的一种集合，但是可以动态的改变集合元素的数量
<code>Stack</code>	一种以后进先出 (Last In First Out, LIFO) 原则工作的集合，集合中最后增加的一个元素会最先从集合中移出
<code>Queue</code>	一种以先进先出 (First In First Out, FIFO) 原则工作的集合，集合中最先进入的一个元素会最先从集合中移出
<code>HashTable</code>	键 (key) / 值 (value) 对的集合，这些键/值对根据键的哈希代码进行组织
<code>SortedList</code>	按照键值进行排序的键/值对的集合，集合中的元素即可以按照键值访问，也可以按照索引访问

上述这些类都实现了 `ICollection` 接口，在 `ICollection` 接口中有 3 个属性和 1 个方法，分别是：

- ☐ `Count` 属性，能够返回当前集合中元素的数量（与数组的 `Length` 属性有些相似）。
- ☐ `IsSynchronized` 属性，能够根据访问集合的线程是否安全返回布尔类型的值。
- ☐ `SyncRoot` 属性，可以返回同步访问集合的对象。
- ☐ `CopyTo(Array array, int index)` 方法，调用该方法可以将集合中特定索引之后的元素复制到指定数组中。

说明：所有的集合类都实现了 `IEnumerable` 接口，这样就可以使用 `foreach` 循环对集合中的元素进行遍历。

10.3.1 用 `ArrayList` 集合保存不同类型数据

`System.Collections.ArrayList` 类与数组比较相似，但是在该类中可以同时存放不同数据类型的元素。在使用 `ArrayList` 类的时候，不需要对创建集合的大小进行声明（在使用数组的时候需要事先声明）。`ArrayList` 集合的大小是可以动态改变的，当集合内的元素数量发生变化时，集合的大小也随之改变。在 `ArrayList` 类的内部定义了一个数组，数组的大小按照默认值 `Capacity` 进行初始化。当 `ArrayList` 集合的元素增加或减少的时候，就修改 `ArrayList` 类内部数组的大小，并根据大小创建一个新数组，再将原有数组中的元素复制到新数组中。`ArrayList` 集合的大小表示当前集合中所有元素的数量。

可以用如下代码创建一个 `ArrayList` 类型的集合：

```
ArrayList list = new ArrayList();
```

在创建 `ArrayList` 类型集合时，也可以通过传入参数对其表示内部数组初始化大小的 `Capacity` 变量进行赋值。代码设置如下：

```
ArrayList list = new ArrayList(20);
```

此外，还可以在创建 `ArrayList` 类型集合的时候，将其他集合的元素通过参数传入集合中，创建方法如下：

```
ArrayList list = new ArrayList(otherCollection);
```

在创建 `ArrayList` 类型的集合之后，可以使用 `Add()` 方法增加集合中的元素。`Add()` 方法的参数是 `object` 类型，这样就可以添加任何类型的元素。在添加元素的时候，可以进行如下操作：

```
list.Add(45);
list.Add(63);
list.Add(12);
```

经过如上操作之后，就可以在 `ArrayList` 类型的 `list` 集合中添加 3 个元素。这时用 `foreach` 循环对 `list` 集合进行遍历，就可以得到集合中的元素，代码如下：

```
//声明使用的命名空间
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _10._3._1
{
    //定义 Program 类
    class Program
```



```

{
    static void Main(string[] args)
    {
        //创建 ArrayList 类的对象
        ArrayList list = new ArrayList();
        //调用 list 对象的方法添加数据
        list.Add(45);
        list.Add(63);
        list.Add(12);
        //输出 list 对象中的数据
        foreach (int num in list)
        {
            Console.WriteLine(num);
        }
    }
}

```

程序执行后将输出集合中的元素，结果如下：

```

45
63
12

```

在 ArrayList 类中还实现了索引的属性，该属性允许使用“[]”操作符访问集合中的元素，与数组中的索引操作类似。使用该属性，可以将以上程序改写成：

```

static void Main( )
{
    //创建 ArrayList 类的对象
    ArrayList list = new ArrayList();
    //调用 list 对象的方法添加数据
    list.Add(45);
    list.Add(63);
    list.Add(12);
    //输出 list 对象中的数据
    for (int i = 0; i < list.Count; i++)
    {
        Console.WriteLine(list[i]);
    }
}

```

上述程序执行后将与之前的程序具有相同的输出结果。该程序中使用了 ArrayList 集合的 Count 属性，用于获取集合中元素的数量。


说明：该属性是 ArrayList 类从 ICollection 接口继承来的，其他的集合类也有继承该接口的情况。

表 10.2 列出了 ArrayList 类中一些重要的属性和方法。


表 10.2 ArrayList类中的属性和方法

属性或方法	描 述
Capacity	获取或者设置 ArrayList 集合中可以包含的元素的数量
Count	获取 ArrayList 集合中元素的数量
Add(object)	在集合的末尾添加一个新的元素, 该元素是 object 类型的
Remove(object)	从集合中移除一个元素
RemoveAt(int)	移除集合指定索引处的元素
Insert(int,object)	在指定的索引处添加一个集合元素
Clear()	移除集合中所有的元素
Contains(object)	返回布尔类型的值表明集合中是否包含指定的元素
CopyTo()	将集合中的元素复制到一个指定的数组中。该方法是一个重写后的方法, 使用的时候可以指定元素复制的范围, 以及从哪个元素开始复制
IndexOf(object)	返回指定元素在 ArrayList 集合中第 1 次出现时的索引值, 索引值是从 0 开始计数的, 即集合第一个元素的索引值是 0。如果在集合中不存在指定的元素, 则返回-1
LastIndexOf(object)	返回指定元素在 ArrayList 集合中最后一次出现时的索引值
ToArray()	返回一个 object 类型的数组, 该数组中包含 ArrayList 集合中所有的元素
TrimToSize()	设置 Capacity 属性的值为当前 ArrayList 集合中实际的元素数量

10.3.2 用 Stack 集合处理栈

System.Collections.Stack 类是一种提供元素访问控制的集合。Stack（栈集合）是按照后进先出的原则（Last In First Out, LIFO）工作的, 也就是说最后一个进入集合的元素将最先从集合中移出。

在对 Stack 类进行介绍之前, 需要先说明栈的概念。栈和堆是计算机中两个被普遍使用的数据结构, 并且它们都是由硬件和软件同时实现的。栈的工作原则是先进先出原则, 即最后进栈的变量必须最先出栈。向栈中添加变量的过程叫做进栈（也称为 Push）, 而从栈中获取变量的过程叫做出栈（也称为 Pop）。还有对栈的 Peek 操作, 该操作获取最后一个进栈变量的值, 但是该元素仍保留在栈中而不出栈。

 **说明:** Stack（栈集合）是按照后进先出的原则（Last In First Out, LIFO）工作的。

在.NET 环境中, 通过 System.Collections.Stack 类实现栈的功能。Stack 类能够以 ArrayList 类一样的方式进行实例化, 示例如下:

```
Stack stack = new Stack();
```

Stack 类的默认构造函数将自动初始化一个空的栈, 而下面的一种创建方式则会给新创建的栈规定栈的容量。

```
Stack stack = new Stack(12);
```

与 ArrayList 类型的集合类似, Stack 类也能以其他集合为参数进行实例化操作, 这样就会直接将其他集合的元素加入栈中。示例如下:


```
Stack stack = new Stack(otherCollection);
```

可以通过调用 `Push()` 方法执行元素的进栈操作，具体操作如下：

```
stack.Push(2);
stack.Push(4);
stack.Push(6);
```

与上例类似的是，可以使用 `Pop()` 方法重新获得栈中的元素。下面的程序给出了如何将多个元素放入栈中，并将它们逐个地从栈中取出。

```
//声明使用的命名空间
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _10._3._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //创建 Stack 类的对象
            Stack stack = new Stack();
            //向栈中添加数据
            stack.Push(2);
            stack.Push(4);
            stack.Push(6);
            //输出栈中的数据
            while (stack.Count != 0)
            {
                Console.WriteLine(stack.Pop());
            }
        }
    }
}
```

在程序中使用了 `While` 循环用于获取栈中所有的元素，这是由栈的特性决定的。这里需要注意，对栈执行 `Pop` 操作不仅获取了最后一个进栈元素的值，而且对该元素执行了出栈操作，因此在执行 `Pop` 操作之后栈的 `Count` 值将会减 1。

程序在执行后将输出如下结果：

```
6
4
2
```

如果只是希望获取最后一个进栈元素的值，而不希望对该元素执行出栈操作，那么可以使用 `Peek()` 方法。下面的程序就给出了该方法的一个示例。

```
//声明使用的命名空间
using System;
```

```

using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _10._3._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //创建 Stack 类的对象
            Stack stack = new Stack();
            stack.Push(2);
            stack.Push(4);
            stack.Push(6);
            //获取栈中的最后一个数据
            Console.WriteLine("在执行 Peek 操作之前栈中元素总数是:
{0}", stack.Count);
            Console.WriteLine("最后一个进栈的元素是: {0}", stack.Peek());
            Console.WriteLine("在执行 Peek 操作之后栈中元素总数是:
{0}", stack.Count);
        }
    }
}

```

在上面的程序中，首先执行了 3 次进栈操作，然后使用 Peek() 方法获取了最后一个进栈元素的值。程序执行后将输出如下结果：

```


在执行 Peek 操作之前栈中元素总数是: 3
最后一个进栈的元素是: 6
在执行 Peek 操作之后栈中元素总数是: 3

```

从输出结果可以看出，Peek() 方法执行后只获取了栈中最后一个元素的值，而没有使其出栈。这点与 Pop() 方法不同，在程序设计的时候可以灵活使用。

10.3.3 用 Queue 集合处理队列

集合（Queue）是按照先进先出（First In First Out, FIFO）原则工作的，意味着最先进入队列的元素会最先移出队列。向队列中添加元素的过程被称为“Enqueue”，而从队列中移出元素的过程则称为“Dequeue”。与栈操作相似，队列中也有相应的 Peek 操作，执行该操作后只获得队列首元素的值，而不会将该元素移出队列。

说明：集合（Queue）是按照先进先出（First In First Out, FIFO）原则工作的。

在 .NET 环境中，System.Collections.Queue 类提供了队列的功能。Queue 类的实例化操作与 ArrayList 和 Stack 类十分相似，有以下 3 种操作：

```

//创建一个空的队列
Queue queue = new Queue();

```



```
//创建一个具有元素数量限制的队列
Queue queue = new Queue(16);
//创建一个使用其他集合元素进行初始化的队列
Queue queue = new Queue(otherCollection);
```


下面的程序是对 Queue 类的应用示例:

```
//声明使用的命名空间
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _10._3._3
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //创建 Queue 类的对象
            Queue queue = new Queue();
            //向集合中添加数据
            queue.Enqueue(2);
            queue.Enqueue(4);
            queue.Enqueue(6);
            //输出集合中的数据
            while (queue.Count != 0)
            {
                Console.WriteLine(queue.Dequeue());
            }
        }
    }
}
```

在程序中向队列中添加了 3 个元素, 并且使用 while 循环逐一将这些元素从队列中读出。程序在执行后将输出如下结果:

```
2
4
6
```

从程序的输出结果可以看出, 队列移出元素的顺序与进入队列的顺序是一致的。

说明: Queue 类的其他方法与 ArrayList 和 Stack 类中的方法十分相似。

10.4 键-值对应形式的字典类型集合

字典类型是集合中一种以键-值对应的形式存储数据的集合类型, 集合中的每一个元素

(值, 也称为 value) 都有一个与其对应的键值 (key)。集合中的每一个键值都是唯一的, 也就是不可能有多个键值具有相同的值。这就像使用的字典一样, 字典中的每一个词都有与其对应的解释, 这也是字典类型集合名字的由来。

在 System.Collections 命名空间中有两个经常使用的字典类型的集合, 分别是 Hashtable 类和 SortedList 类, 下面将分别进行介绍。

10.4.1 Hashtable 类存储方式

Hashtable 类型的集合中按照键-值 (key-value) 对应的形式对集合的元素进行存储, 在 Hashtable 集合中的每一个元素 (值) 都有唯一的用于识别的键值。在 Hashtable 集合中, 对元素和其对应的键值都是以 object 类型进行保存的。大多数情况下, 都是使用 String 类作为 Hashtable 集合的键值类型。不过, 键值也可以是其他的数据类型。但是在选择一种数据类型作为 Hashtable 集合的键值时, 必须重写这些类型从 object 类中继承的 Equals() 和 GetHashCode() 方法。需要注意以下几点:

- (1) Equals() 方法用于检查实例与默认引用的相等性。
- (2) GetHashCode() 方法对相似的类的实例将返回相同的整数值。
- (3) GetHashCode() 方法返回的整形数值将均匀分布在最小值和最大值之间。

下面将介绍对 Hashtable 类型集合的基本操作。

10.4.2 创建 Hashtable 集合的方法

在基本类库中提供的 string 类型以及其他一些类型都考虑到了对 Hashtable 集合的支持, 这些类型非常适合作为 Hashtable 集合或者其他字典集合的键值。可以有多种方法用于创建 Hashtable 类型集合的实例, 最简单的方法如下:

```
Hashtable ht = new Hashtable();
```

这种方法使用了默认的没有参数的构造方法。也可以在构造 Hashtable 实例的时候限定其元素的上限值, 例如:

```
Hashtable ht = new Hashtable(20);
```

与 ArrayList 等类相似, Hashtable 类也可以使用其他集合作为构造函数的参数, 从而对创建的 Hashtable 集合进行元素的初始化操作。代码如下:

```
Hashtable ht = new Hashtable(otherCollection);
```

10.4.3 用 Add 方法向 Hashtable 集合中添加元素

在创建 Hashtable 类型的集合之后, 可以使用 Add() 方法向创建的集合中添加新的元素:

```
ht.Add("st01", "Beijing");  
ht.Add("st02", "Shanghai");
```




```
ht.Add("st03", "Shenzhen");
```

10.4.4 从 Hashtable 集合中获取元素的方法

在 10.4.3 节的程序中，已经在 Hashtable 集合中添加了 3 个元素及其相应的键值。在此之后，就可以通过使用键值获得相应的元素。示例代码如下：

```
Console.WriteLine("Hashtable 集合的大小是: {0}", ht.Count);
Console.WriteLine("键值是"st01"的元素是: {0}", ht["st01"]);
Console.WriteLine("Hashtable 集合的大小是: {0}", ht.Count);
```

说明：在程序中，使用了索引（“[]”操作）用于从 Hashtable 集合中获取集合元素。使用这种方式获取集合元素之后，并不会移除集合中的元素，而只是返回指定键值对应的元素对象。因此，在执行元素的获取操作之后，Hashtable 集合的大小不会发生改变。

程序执行后的输出结果如下：

```
Hashtable 集合的大小是: 3
键值是 "st01" 的元素是: Beijing
Hashtable 集合的大小是: 3
```

10.4.5 用 Remove 方法移除 Hashtable 集合中的元素

可以使用 Remove()方法移除 Hashtable 集合中指定的元素。在调用 Remove()方法的时候，使用元素对应的键值作为方法的参数。以下程序给出了 Remove()方法的示例。

```
//声明使用的命名空间
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _10._4._5
{
    class Program
    {
        static void Main( )
        {
            //创建 Hashtable 类的对象
            Hashtable ht = new Hashtable(20);
            //向集合中添加数据
            ht.Add("st01", "Beijing");
            ht.Add("st02", "Shanghai");
            ht.Add("st03", "Shenzhen");
            //从集合中移除数据
            Console.WriteLine("Hashtable 集合的大小是: {0}", ht.Count);
```

```

        Console.WriteLine("从集合中移除键值为\"st01\"的元素");
        ht.Remove("st01");
        Console.WriteLine("Hashtable 集合的大小是: {0}", ht.Count);
    }
}

```

程序执行后，将输出如下结果：

```

Hashtable 集合的大小是: 3
从集合中移除键值为\"st01\"的元素
Hashtable 集合的大小是: 2

```

10.4.6 获取 Hashtable 集合的元素与键值

Hashtable 集合中的所有元素和键值都可以通过使用 Values 和 Keys 两个属性获得，下面的程序给出了对 Values 和 Keys 两个属性的应用，示例如下：

```

static void Main( )
{
    //创建 Hashtable 类的对象
    Hashtable ht = new Hashtable(20);
    //向 ht 中添加数据
    ht.Add("st01", "Beijing");
    ht.Add("st02", "Shanghai");
    ht.Add("st03", "Shenzhen");
    //输出 ht 中的键值
    Console.WriteLine("输出 Hashtable 集合中的键值: ");
    foreach(string key in ht.Keys)
    {
        Console.WriteLine(key);
    }
    //输出 ht 中的元素
    Console.WriteLine("\n 输出 Hashtable 集合中的元素: ");
    foreach(string value in ht.Values)
    {
        Console.WriteLine(value);
    }
}

```

程序执行后的输出结果如下：

```

输出 Hashtable 集合中的键值:
st01
st02
st03

输出 Hashtable 集合中的元素:
Beijing
Shanghai
Shenzhen

```


10.4.7 在 Hashtable 集合中检索元素

在使用 Hashtable 集合的时候, 可以使用 ContainsKey()方法和 ContainsValue()方法用于检查集合中是否存在指定的键值或者元素。这两个方法在调用后都会返回布尔类型的返回值, 用以表示集合中是否存在所查找的内容。

```
static void Main( )
{
    Hashtable ht = new Hashtable(20);
    ht.Add("st01", "Beijing");
    ht.Add("st02", "Shanghai");
    ht.Add("st03", "Shenzhen");
    Console.WriteLine(ht.ContainsKey("st01"));
    Console.WriteLine(ht.ContainsKey("st06"));
    Console.WriteLine(ht.ContainsValue("Beijing"));
}
```

执行后的输出结果是:

```
True
False
True
```

从程序的执行结果可以看出, ContainsKey()方法和 ContainsValue()方法表明了指定的键值与元素在集合中是否存在。

10.4.8 SortedList 类与 Hashtable 类的区别

SortedList 类与 Hashtable 类非常相似。这两个类之间的不同处在于, SortedList 集合中的元素根据其对应键值进行了排序, 而 Hashtable 集合中的元素没有进行这样的排序。使用 SortedList 类型集合的一个好处就是, 可以对集合中的元素使用整型数值进行索引, 就像对数组进行的索引操作一样。在使用 SortedList 集合时, 也可以使用自定义的类当作集合元素的键值。但是, 在选用自定义的类当作集合键值的时候, 除了需要实现介绍 Hashtable 集合时所具备的一些条件外, 还必须确认这些类实现了 IComparable 接口。


10.4.9 SortedList 类常用方法

在 IComparable 接口中只有一个成员方法, 就是 CompareTo()方法, 该方法的声明如下:

```
int CompareTo(object obj);
```

从上面的声明可以看出, CompareTo()方法的参数是 object 类型的对象, 并且方法调用后将返回一个整型的数值, 用来表示当前的对象与参数所提供的对象间的关系, 具体表现为:

- (1) 返回值是 0，表示当前对象与参数所提供的对象相等。
- (2) 返回值大于 0，表示当前对象大于参数所提供的对象。
- (3) 返回值小于 0，表示当前对象小于参数所提供的对象。

 **说明：**String 类和其他一些简单的数据类型（如 int 类型等）都提供了对 ICompare 接口的实现，因此这些类都可以被用做 SortedList 类型集合的索引。

SortedList 类提供与 Hashtable 类相似的构造函数，下面的示例代码说明了 SortedList 类的不同构造函数的使用。

```
//创建一个空的 SortedList 集合
SortedList sl = new SortedList();
//创建一个具有元素数量限制的 SortedList 集合
SortedList sl = new SortedList(16);
//创建一个使用其他集合元素进行初始化的 SortedList 集合
SortedList sl = new SortedList(otherCollection);
```

如表 10.3 所示，列出了 SortedList 类中常用的属性和方法。

表 10.3 SortedList 类中的属性和方法

属性或方法	描 述
Count	得到 SortedList 集合中当前包含元素的数量
Keys	以数组形式返回 SortedList 集合中所有键值 (Key)
Values	以数组形式返回 SortedList 集合中所有元素值 (Value)
Add(object key, object value)	向 SortedList 集合中添加新的元素 (键-值对应的形式)
GetKey(int index)	返回指定索引的键值
GetByIndex(int index)	返回指定索引的元素值
IndexOfKey(object key)	返回指定键值在 SortedList 集合中的索引值, 索引值是从 0 开始计数的, 即集合第一个元素的索引值是 0
IndexOfValue(object key)	返回指定元素值在 SortedList 集合中的索引值
Remove(object key)	从 SortedList 集合中移除指定键值的元素
RemoveAt(int index)	从 SortedList 集合中移除指定索引值的元素
Clear()	从 SortedList 集合中移除所有的元素
ContainsKey(object key)	返回布尔类型的值表明 SortedList 集合中是否包含指定键值的元素
ContainsValue(object value)	返回布尔类型的值表明 SortedList 集合中是否包含指定元素值的元素

下面的程序给出了对 SortedList 集合应用的示例：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _10._4._9
{
    //定义 Program 类
    class Program
```



```

{
    static void Main(string[] args)
    {
        //创建 SortedList 类的对象
        SortedList sl = new SortedList( );
        //向 st 中添加数据
        sl.Add(32, "Java");
        sl.Add(16, "C#");
        sl.Add(3, "VB.NET");
        sl.Add(12, "C++");
        //输出 st 中的数据
        Console.WriteLine("集合中元素存放顺序是: ");
        Console.WriteLine("\t Key \t\t Value");
        Console.WriteLine("\t === \t\t =====");
        for(int i = 0; i < sl.Count; i++)
        {
            Console.WriteLine("\t {0} \t\t {1}", sl.GetKey(i),
                               sl.GetByIndex(i));
        }
    }
}

```

在上述示例程序中，SortedList 集合以字符串的形式保存了不同程序语言的名称，这些集合元素使用整数类型的键值。在程序中，使用 for 循环获取 SortedList 集合中的键值和元素值。因为 SortedList 集合对集合内的元素自动进行了排序，所以集合元素都是以排序后的顺序进行存储的。因此，当使用 GetKey() 或者 GetByIndex() 方法获取集合中元素的时候，得到的都是有序元素。关于这点可以从以下的程序输出结果中看出：

集合中元素存放顺序是：

```

Key  Value
===  =====
3    VB.NET
12   C++
16   C#
32   Java

```

10.5 本章总结

本章主要介绍了.NET 中的多维数组和集合的概念，虽然这两个概念都比较简单，但是对多维数组和集合的有效利用，能够提高程序的可用性和健壮性。

数组是一系列相同数据类型元素的集合。在 C# 中，数组的元素可以是对象，也可以是引用类型，所以数组是在堆上存储的。多维数组可以看做“数组的数组”。也就是说，多维数组中的一个元素可以是其他的数组。在 C# 中，多维数组可以分成两种类型：

- (1) 矩形数组（数组的每个行元素都包含相同个数的列元素）。
- (2) 变长数组（数组的每个行元素可以包含不同数量的列元素）。

多维数组虽然给程序的设计带来了一定的便利性，但是多维数组中只能存放一个类型的对象，并且数组在定义的时候就必须限定其大小。所以，才在 C# 中引入了集合的概念，集合解决了数组提到的上述问题，而且还增加了对其中元素的插入和查找等操作，更大地提高了集合的可用性。

本章中主要介绍了 .NET 环境中几个典型的集合类型，分别是 `ArrayList` 类、`Stack` 类、`Queue` 类，以及字典类型的集合。读者需要明白这些集合的特点和使用时要注意的要点，正确地使用集合将给程序的设计带来极大的便利性。

10.6 实战练习

1. 在 Visual Studio 2010 中创建一个控制台应用程序，定义一个二维数组，使用 `for` 循环语句，从键盘上输入 10 位员工的姓名和当月工资，分别求出最高工资和最低工资，并且求出 10 位员工的平均工资。

2. 在 Visual Studio 2010 中创建一个控制台应用程序，使用 `ArrayList` 保存公司部门名称的集合，然后使用 `foreach` 循环输出所有部门名称。

3. 在 Visual Studio 2010 中创建一个控制台应用程序，用 `Hashtable` 类来保存员工的姓名和工资，计算出员工的平均工资，最后输出员工平均工资、所有员工姓名和工资。

第 11 章 代理和事件

本章将会对 C# 中代理和事件的概念进行重点介绍。代理和事件是 C# 中的重要内容，在这一章中将分为 3 个主要部分进行说明。首先，将阐明代理的基本思想，以及在 C# 中如何正确地使用代理。其次，会进一步深入介绍代理的使用方法，讲解多重代理的使用及其重要性。本章最后将对 C# 中的事件概念进行说明，并介绍通过代理实现的 C# 事件处理机制。

11.1 代 理

11.1.1 什么是代理

简单地说，C# 中的代理就是支持对方法的引用。有些书中，也将代理称为“代表”或者“委托”。

在 C# 中存在多种引用，到目前为止，使用最多的是对象的引用。例如下述代码中，st 就是 Stack 类的对象的引用。

```
Stack st = new Stack( );
```

总的来说，引用具有以下两个属性：

- 引用的类型，也就是引用可以指向的对象的类型。
- 引用指向的实际对象。

代理与对象的引用十分相似，只是代理是被用来指向某个方法而不是对象。代理的类型是某个方法的类型或者方法标识。所以，通常代理具有 3 个属性：

- 代理所指向方法的类型或者标识。
- 能够被用于指向某个方法的代理的引用。
- 代理所指向的实际的方法。

如图 11.1 所示，给出了代理与对象引用的比较。



图 11.1 代理与对象引用的比较图

说明：C#中代理的概念与 C++中方法的指针是比较相似的。

11.1.2 代理所指向方法的类型和标识

在使用一个代理之前，需要明确代理所指向方法的类型和它的方法标识。在这里，方法的标识包括方法的返回类型以及方法需要传入参数的类型。例如下述代码：

```
int someMethod(string [] args)
```

上面的这个方法标识，与 C#中经常使用的 `Main()` 方法的标识是一致的，示例代码如下：

```
int Main(string [] args)
{
    ...//方法处理过程
}
```

再以一个 `Add()` 方法为例，它的方法定义如下：

```
int Add(int a, int b)
{
    return a + b;
}
```

该 `Add()` 方法的标识是：

```
int aMethod(int p, int q);
```

这个标识同样可以作为其他相似方法的标识，例如下面的 `Subtract()` 方法：

```
int Subtract(int c, int d)
{
    return c - d;
}
```

从上面的两个方法的定义可以看出，方法名并不是方法标识的组成部分。实际上，方法标识是指涉及方法的返回值和它的参数。

注意：此处需要区别在第9章节中提到的方法标记的概念，方法标记是与方法名相关的。

在使用的时候，可以使用关键字 `delegate` 来声明一种类型的代理，例如：

```
delegate int MyDelegate(int p, int q);
```

在上面的例子中定义了一个名为 `MyDelegate` 的代理，该代理类型的引用可以用来指向所有返回值为 `int` 类型，并且具有两个 `int` 类型参数的方法。在 11.1.3 节中将对代理的引用进行介绍。

11.1.3 代理引用的声明和使用

在定义了一种代理类型之后，可以使用该代理类型的引用指向符合代理所定义标识的实际方法。代理引用的声明与对象引用的声明过程十分相似，下述代码就声明了一个

MyDelegate 类型的代理引用。

```
MyDelegate arithMethod;
```

在声明 arithMethod 引用之后, 就可以使用它指向任何符合 MyDelegate 类型代理所定义标识的方法。具体的过程是将符合实际方法的方法名作为参数, 通过相应的代理传递给类型代理的引用。示例如下:

```
arithMethod = new MyDelegate(Add);
```

这样, 通过上述的操作, arithMethod 代理引用就被设置成指向 Add() 方法。此外, 上述两个过程可以被整合成一个操作:

```
MyDelegate arithMethod = new MyDelegate(Add);
```

可以看出, 这与对象引用的声明和使用是一样的。同样, 也可以使用代理引用调用所指向的实际方法。例如, 可以使用 arithMethod 代理引用调用 Add() 方法:

```
int r = arithMethod(3, 4);
```

通过如上的操作, 就可以计算 3 和 4 相加后的值, 并传递给整数变量 r。在实际的程序设计过程中, 可以这样使用代理, 代码如下:

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _11._1._3
{
    //定义 Sample 类
    class Sample
    {
        //定义 MyDelegate 代理
        delegate int MyDelegate(int p, int q);
        //定义符合 MyDelegate 代理类型的方法
        static int Add(int a, int b)
        {
            return a + b;
        }
        static void Main( )
        {
            //创建一个指向 Add( ) 方法的代理的引用
            MyDelegate arithMethod = new MyDelegate(Add);
            //通过代理引用调用 Add( ) 方法
            int r = arithMethod(3, 4);
            Console.WriteLine("3 和 4 相加的结果是: {0}", r);
        }
    }
}
```

程序执行后的输出结果是:

```
3 和 4 相加的结果是: 7
```

在明白了可以使用代理的引用调用实际方法之后，就可以使用代理进行更灵活的操作。对上述程序进行修改，使用户能够选择将要执行的算术操作，示例代码如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _11._1._3
{
    //定义 Program 类
    class Program
    {
        //定义 MyDelegate 代理
        delegate int MyDelegate(int p, int q);
        //定义程序的 Main( ) 方法
        static void Main( )
        {
            //创建 MyDelegate 类型的引用
            MyDelegate arithMethod = null;
            string myOperate = null;
            //打印输出提示信息
            Console.WriteLine("选择对 3 和 4 执行的算术操作: ");
            Console.WriteLine("按"+"键执行加法操作: ");
            Console.WriteLine("按"-"键执行减法操作: ");
            Console.WriteLine("按"m"键求两个数的最大值。");
            char choice = (char) Console.Read( );
            //根据用户输入进行不同处理
            switch(choice)
            {
                case '+':

                    arithMethod = new MyDelegate(Add);
                    myOperate = "加法";
                    break;
                case '-':
                    //使 arithMethod 引用指向 Subtract( ) 方法
                    arithMethod = new MyDelegate(Subtract);
                    myOperate = "减法";
                    break;
                case 'm':
                    //使 arithMethod 引用指向 Max( ) 方法
                    arithMethod = new MyDelegate(Max);
                    myOperate = "求最大值";
                    break;
                default :
                    //默认使 arithMethod 引用指向 Add( ) 方法
                    arithMethod = new MyDelegate(Add);
                    myOperate = "加法";
                    break;
            }
        }
    }
}
```



```

    }
    //通过 arithMethod 引用调用相应方法
    int r = arithMethod(3, 4);
    Console.WriteLine("\n对 3 和 4 执行{0}操作后的结果是:{1}", myOperate, r);
}
//定义符合 MyDelegate 代理类型的 Add( ) 方法
static int Add(int a, int b)
{
    return a + b;
}
//定义符合 MyDelegate 代理类型的 Subtract( ) 方法
static int Subtract(int a, int b)
{
    return a - b;
}
//定义符合 MyDelegate 代理类型的 Max( ) 方法
static int Max(int a, int b)
{
    if(a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
}
}

```

在程序中定义了 3 个具有相同方法标识的方法：Add()、Subtract()和 Max()。而且，还定义了一个 MyDelegate 类型的代理引用 arithDelegate，使用这个引用可以指向所有标识符合的方法。本例中的 arithDelegate 代理引用可以指向用户选择的方法，并执行该方法的操作。

程序在执行后将输出如下的结果：

```

选择对 3 和 4 执行的算术操作：
按"+"键执行加法操作；
按"-"键执行减法操作；
按"m"键求两个数的最大值。


```

```

对 3 和 4 执行减法操作后的结果是：-1

```


可以看出，程序执行的是减法运算。这是因为在程序执行的时候按了“-”键，从而代理引用指向了 Subtract()方法。

 **注意：**同一个代理引用可以指向不同的方法，只要这些方法具有与代理定义相同的方法标识。

11.1.4 .NET Framework 中的代理

在 C#中以类的形式对代理进行了实现，并且设定了相应的关键字。实际上，在.NET 中使用代理的时候，更多的表现为一种引用，而不是类的实例。但是，需要明白的是，所有的代理都继承了 `System.Delegate` 类。因此，理论上讲，之前介绍代理时说的“代理是对方法的引用”并不十分准确，一个代理其实是继承 `System.Delegate` 类后所形成的新引用类型，使用该类型的实例可以调用与代理定义方法标识吻合的方法。需要读者理解的是，定义一个新的代理就是创建了 `System.Delegate` 类的一个子类，所以不能在某个方法的内部定义代理（对其他类也有这个限定）。这就是在之前的示例中，始终都把代理定义在 `Main()` 方法外的原因，示例代码如下：

```
class Sample
{
    delegate int MyDelegate(int p, int q);
    static void Main( )
    {
        MyDelegate arithMethod = null;
        ...
    }
}
```

 注意：不能在某个方法的内部定义代理。

11.1.5 代理用作方法的参数

对象的引用可以被其他的对象使用，与此相似，代理虽然是某个方法的引用，但是也可以被其他方法调用。这种调用是通过将代理作为参数传递给方法实现的，如下面的方法声明所示：

```
static void PerformArithOperation(int a, int b, MyDelegate arithOperation)
{
    int r = arithOperation(a, b);
    Console.WriteLine("\n 对 3 和 4 进行算术操作后的结果是: {0} ", r);
}
```

`PerformArithOperation()` 方法将 `MyDelegate` 类型的代理作为参数，方法被调用后，一个 `MyDelegate` 代理的实例就会被传到方法内部，这样在方法内就可以执行代理所引用的方法，完成特定的操作。下面的程序就是使用代理作为方法参数的完整的示例：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _11._1._5
{
    ...
}
```



```

//定义 Sample 类
class Sample
{
    //定义 MyDelegate 代理
    delegate int MyDelegate(int p, int q);
    //定义程序的 Main( ) 方法
    static void Main( )
    {
        //创建 MyDelegate 类型的引用
        MyDelegate arithMethod = null;
        //打印输出提示信息
        Console.WriteLine("选择对 3 和 4 执行的算术操作;");
        Console.WriteLine("按 “+” 键执行加法操作;");
        Console.WriteLine("按 “-” 键执行减法操作;");
        Console.WriteLine("按 “m” 键求两个数的最大值。");
        char choice = (char) Console.Read( );
        //根据用户输入进行不同的处理
        switch(choice)
        {
            case '+':
                //使 arithMethod 引用指向 Add( ) 方法
                arithMethod = new MyDelegate(Add);
                break;
            case '-':
                //使 arithMethod 引用指向 Subtract( ) 方法
                arithMethod = new MyDelegate(Subtract);
                break;
            case 'm':
                //使 arithMethod 引用指向 Max( ) 方法
                arithMethod = new MyDelegate(Max);
                break;
            default :
                //默认使 arithMethod 引用指向 Add( ) 方法
                arithMethod = new MyDelegate(Add);
                break;
        }
        PerformArithOperation(3, 4, arithMethod);
    }
    //使用 MyDelegate 类型变量作为方法参数
    static void PerformArithOperation(int a, int b, MyDelegate
    arithOperation)
    {
        int r = arithOperation(a, b);
        Console.WriteLine("\n 对 3 和 4 执行算术操作后的结果是: {0}", r);
    }
    //定义符合 MyDelegate 类型的 Add( ) 方法
    static int Add(int a, int b)
    {
        return a + b;
    }
    //定义符合 MyDelegate 类型的 Subtract( ) 方法
    static int Subtract(int a, int b)

```

```


    {
        return a - b;
    }
    //定义符合 MyDelegate 类型的 Max( ) 方法
    static int Max(int a, int b)
    {
        if(a > b)
        {
            return a;
        }
        else
        {
            return b;
        }
    }
}

```

11.1.6 了解多重代理

在 C# 中，可以使用一个代理同时指向多个符合方法标识的方法，这就是所谓的多重代理。多重代理是 .NET 中代理的一个重要特点，它可以使程序在设计的时候变得更加方便和灵活。实质上，多重代理是对 `System.MulticastDelegate` 类的继承和实现，该类也是 `System.Delegate` 类的一个子类。

多重代理在使用的时候有一定的限制，就是多重代理所定义的方法标识的返回值必须是 `void` 类型的，也就是说方法不具有返回值。这点在使用多重代理的时候必须十分注意。需要限定多重代理方法标识返回值的原因是，由于多重代理可能会指向多个方法，如果每个方法都有自己的返回值，那么在代理执行后会同时返回多个值，这与一个代理（方法）只能有一个返回值的原则是相互矛盾的，所以在定义多重代理方法标识的时候，必须保证其返回值是 `void` 类型的。

 **注意：**多重代理的方法标识的返回值必须是 `void` 类型的。

11.1.7 多重代理的实现方法

多重代理在定义的时候与前面介绍的代理基本一样，只是在设置代理所指向方法的时候有所不同，而且必须保证其指向的方法没有返回值。

在使用多重代理的时候，首先需要定义代理可以引用的方法的标识，示例代码如下：

```
delegate void MyMulticastDelegate(int p, int q);
```

接下来就可以设定多重代理指向的方法，如果一个多重代理引用多于一个方法，那么就需要使用“+”操作符完成新方法的添加，示例代码如下：

```
MyMulticastDelegate arithMethod = null;
arithMethod = new MyMulticastDelegate(Add);
```



```
arithMethod += new MyMulticastDelegate(Subtract);
arithMethod += new MyMulticastDelegate(Max);
```

在完成如上的多重代理定义后，就可以使用多重代理进行方法的调用。多重代理的调用方式与普通代理相同，不同的是如果使用多重代理后，那么多重代理所指向的所有方法都会被执行。具体示例如下：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _11._1._7
{
    //定义 Sample 类
    class Sample
    {
        //定义 MyMulticastDelegate 代理
        delegate void MyMulticastDelegate(int p, int q);
        //定义程序的 Main( ) 方法
        static void Main( )
        {
            //创建 MyMulticastDelegate 类型的引用
            MyMulticastDelegate arithMethod = null;
            //使 arithMethod 引用指向不同的方法
            arithMethod = new MyMulticastDelegate(Add);
            arithMethod += new MyMulticastDelegate(Subtract);
            arithMethod += new MyMulticastDelegate(Max);
            //通过 arithMethod 引用调用方法
            arithMethod(3, 4);
        }
        //定义符合 MyMulticastDelegate 类型的 Add( ) 方法
        static void Add(int a, int b)
        {
            Console.WriteLine("3 和 4 的和是: {0}", a + b);
        }
        //定义符合 MyMulticastDelegate 类型的 Subtract( ) 方法
        static void Subtract(int a, int b)
        {
            Console.WriteLine("3 和 4 的差是: {0}", a - b);
        }
        //定义符合 MyMulticastDelegate 类型的 Max( ) 方法
        static void Max(int a, int b)
        {
            if(a > b)
            {
                Console.WriteLine("3 和 4 中的最大值是: {0}", a);
            }
            else
            {
                Console.WriteLine("3 和 4 中的最大值是: {0}", b);
            }
        }
    }
}
```

```

    }
}

```

可以看出, 在上面的程序中已经对 `Add()`、`Subtract()` 和 `Max()` 方法进行了修改, 使这些方法都不再具有返回值, 而是通过在每个方法中打印的方式输出相应的结果。


程序在执行之后将会输出如下结果:

```

3 和 4 的和是: 7
3 和 4 的差是: -1
3 和 4 中的最大值是: 4

```

可见, 多重代理在调用之后, 所有该代理所指向的方法都被调用。多重代理的这个特点在 .NET 中的事件机制中得到应用, 事件的处理方法就是通过多重代理进行调用的。这方面的内容将在 11.2 节中进行介绍。

 **说明:** 多重代理在调用之后, 所有该代理所指向的方法都被调用。

11.1.8 怎样移除多重代理指向的方法

前面已经介绍过, 可以使用 “+=” 操作符添加多重代理所指向的对象, 相应的也可以使用 “-=” 操作符移除当前多重代理所指向的对象。下面的程序给出了相应的说明, 代码如下:

```

static void Main( )
{
    Console.WriteLine("当前多重代理所指向的方法: ");
    //创建 MyMulticastDelegate 类型的引用
    MyMulticastDelegate arithMethod = null;
    arithMethod = new MyMulticastDelegate(Add);
    arithMethod += new MyMulticastDelegate(Subtract);
    arithMethod += new MyMulticastDelegate(Max);
    arithMethod(3, 4);
    Console.WriteLine("\n 从多重代理中移除 Subtract ( ) 方法后, 当前多重代理所指向的方法: ");
    arithMethod -= new MyMulticastDelegate(Subtract);
    arithMethod(3, 4);
}

```

在上面的程序里, 首先使 `MyMulticastDelegate` 多重代理指向了 3 个方法, 然后又从这 3 个方法中移除了 `Subtract()` 方法。通过前后两次的打印输出, 可以看到移除方法前后的区别。

程序执行后将打印输出如下结果:

```

当前多重代理所指向的方法:
3 和 4 的和是: 7
3 和 4 的差是: -1
3 和 4 中的最大值是: 4


```


从多重代理中移除 Subtract () 方法后, 当前多重代理所指向的方法:

3 和 4 的和是: 7

3 和 4 中的最大值是: 4

从上面的结果可以看出, 移除 Subtract() 方法前, 多重代理调用后执行 3 个方法。而移除 Subtract() 方法后, 只执行剩下的两个方法。

 **技巧:** 在进行程序设计的时候, 可以对多重代理进行灵活应用。

11.2 事件和事件处理

事件是应用程序在执行过程中所关注的一些动作, 当这些动作发生的时候程序需要对它们做出响应。事件的概念比较广泛, 所有程序需要进行响应处理的动作都可以称为事件, 例如鼠标的单击、键盘的输入或者计时器的消息都可能是程序关注的事件。

事件机制是以消息为基础的, 当特定的动作发生后会产生相应的消息, 关注该事件的应用程序收到事件发生的消息就会开始指定的处理过程。产生事件的类称为事件触发类, 而处理事件的类则称为事件的接收类, 用于处理特定事件的方法称为事件的处理方法。

11.2.1 C#的事件处理

在.NET 中, 使用多重代理机制对事件进行实现, 也就是说每个事件实例都是多重代理类型的。同代理一样, C#中的事件也是以类的形式实现的, 并且是 C#的一个基础类。可以使用 event 关键字定义一个事件, 事件实现和处理的具体步骤如下所述。

(1) 在所有类的外部为事件定义一个公共访问类型的代理, 该代理为多重代理, 所以代理定义方法标识的返回为 void 类型。通常情况下, 事件的代理定义一般如下:

```
public void EventDelegate(object sender, EventArgs s);
```

(2) 定义一个可以产生事件的类 (即事件触发类), 在该类的内部使用 event 关键字与之前声明的代理共同定义一个公共访问类型的事件。代码如下:

```
public event EventDelegate MyEvent;
```

在产生事件的类中, 可以设计一个事件产生的逻辑。当特定的动作发生或消息到达时, 就会触发所定义的事件。通常, 当发生一个事件的时候, 事件的第一个参数是事件的产生类 (或触发类), 第二个参数是 System.EventArgs 类的一个子类, 该类中包含了一些需要传递给事件处理方法的数据信息。

事件一般都以下面的方式产生:

```
SomeEventArgs someData = new SomeEventArgs (/*必要的参数信息*/);
MyEvent(this, someData);
```

其中, SomeEventArgs 类就是 System.EventArgs 子类, 该类的定义一般如下:

```
class SomeEventArgs : EventArgs
```

```
{
    ... //数据信息或方法
}
```

如果一个事件发生后，不需要向事件处理方法传递任何参数，那么该事件的产生可以如下所示。


```
MyEvent(this, null);
```

(3) 定义一个用于接收事件的类，该类通常是应用程序的主要构成类，用于完成应用程序对特定事件的响应。在事件的接收类中，需要定义一个事件处理方法完成对事件的处理。这个处理方法必须符合在步骤(1)中声明的公共访问类型代理所定义的方法标识，并且通常情况下都会以“On”作为事件处理方法名字的起始字符。以下就是一个事件处理方法的定义：

```
public void OnMyEvent(object sender, EventArgs e)
{
    //事件处理过程
}
```

接下来就可以在事件接收类里定义一个步骤(2)中所定义的事件触发类的实例，并且将当前类中定义的事件处理方法与事件绑定在一起（即将事件触发类中的代理指向事件处理方法）。其过程如下：

```
EventClass eventObj = new EventClass( );
eventObj.MyEvent += new EventDelegate(OnMyEvent);
```

 **说明：**经过上述3个步骤之后，在事件 MyEvent 发生的时候，就会自动地触发 OnMyEvent() 事件处理方法，实现对特定事件的响应。

11.2.2 事件举例——时钟事件

为了能够更清楚地对.NET的事件机制进行说明，本节中以时钟事件为例介绍事件实现和处理过程。Clock Timer（即时钟计时器）每秒都会触发一个事件，使监视时钟计时的类通过事件得到时钟的计时信息。

在实现时钟事件的时候，首先需要为事件定义一个公共访问类型的代理，定义该代理的名字为“TimerEvent”，具体代码如下：

```
public delegate void TimerEvent(object sender, EventArgs e);
```

接着就可以定义产生时钟计时事件的类，如 ClockTimer 类所示。

```
//定义 ClockTimer 类
class ClockTimer
{
    //定义 Timer 事件
    public event TimerEvent Timer;
    public void Start( )
    {
```



```

        for(int i = 0; i < 5; i++)
        {
            Timer(this, null);
            //休眠 1 秒
            Thread.Sleep(1000);
        }
    }
}

```

在 `ClockTimer` 类中包含一个 `Timer` 事件，该事件是 `TimerEvent` 代理类型的。在 `ClockTimer` 类的 `Start()` 方法中，`Timer` 事件每隔一秒都会产生一次，一共产生 5 次 `Timer` 事件。之所以能达到这样的效果，是因为这是通过使用 `System.Threading.Thread` 类的 `Sleep()` 方法实现的。在调用 `Sleep()` 方法后，将根据具体参数挂起当前的线程，完成计时的目的。

在定义了 `ClockTimer` 事件产生类之后，就需要定义一个用于处理该事件的事件接收类，`Sample` 类就是为了达成该目的所定义的，代码如下：

```

//定义 Sample 类
class Sample
{
    static void Main( )
    {
        ClockTimer clockTimer = new ClockTimer( );
        //向 Timer 中添加方法
        clockTimer.Timer += new TimerEvent(OnClockTick);
        clockTimer.Start( );
    }
    //定义符合类型的 OnClockTick( ) 方法
    public static void OnClockTick(object sender, EventArgs e)
    {
        Console.WriteLine("收到时钟事件！");
    }
}

```

在 `Sample` 类中定义了 `OnClockTick()` 事件处理方法，该方法符合 `ClockEvent` 代理定义的方法标识。在 `Sample` 类的 `Main()` 方法中，首先创建了 `ClockTimer` 事件产生类的实例 `clockTimer`，接着将 `OnClockTick` 方法与 `clockTimer` 中的代理进行绑定，使 `clockTimer` 对象中的代理指向 `OnClockTimer` 方法。完成这两步操作之后，在 `Main()` 方法中调用了 `Start()` 方法，从而每隔一秒就触发一次 `Timer` 事件。完整的应用程序如下所示。

```

//声明使用的命名空间
using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _11._2._2
{
    //定义 Sample 类
    class Sample
    {

```

```

//定义程序的 Main( ) 方法
static void Main( )
{
    ClockTimer clockTimer = new ClockTimer( );
    //向 Timer 中添加方法
    clockTimer.Timer += new TimerEvent(OnClockTick);
    clockTimer.Start( );
}
//定义符合类型的 OnClockTick 方法
public static void OnClockTick(object sender, EventArgs e)
{
    Console.WriteLine("收到时钟事件!");
}
//定义事件的代理
public delegate void TimerEvent(object sender, EventArgs e);
//定义 ClockTimer 类
class ClockTimer
{
    //定义 Timer 事件
    public event TimerEvent Timer;
    public void Start( )
    {
        for(int i = 0; i < 5; i++)
        {
            Timer(this, null);
            Thread.Sleep(1000);
        }
    }
}
}

```


由于在程序中使用了 Thread 类的方法，所以在程序的开始声明了 System.Threading 命名空间。

上述程序在执行后，将输出如下结果：

```

收到时钟事件!
收到时钟事件!
收到时钟事件!
收到时钟事件!
收到时钟事件!

```

说明：程序执行后每隔一秒就输出一一次时钟事件发生的消息，共输出了 5 次，符合 Start() 方法中的定义。

11.2.3 多重事件的处理

由于事件在 C# 中是以多重代理的方式进行实现的，所以在使用的时候可以利用一个事件触发多个处理过程，也就是多重事件的处理。

对 11.2.2 节示例中的 `Sample` 类进行修改, 使其在发生时钟计时事件后触发多个事件处理方法, 具体代码如下:

```
//定义 Sample 类
class Sample
{
    static void Main( )
    {
        ClockTimer clockTimer = new ClockTimer( );
        //向 Timer 中添加方法
        clockTimer.Timer += new TimerEvent(OnClockTick);
        clockTimer.Timer += new TimerEvent(OnClockTick2);
        clockTimer.Start( );
    }
    //定义符合类型的 OnClockTick( ) 方法
    public static void OnClockTick(object sender, EventArgs e)
    {
        Console.WriteLine("收到时钟事件!");
    }
    //定义符合类型的 OnClockTick2( ) 方法
    public static void OnClockTick2(object sender, EventArgs e)
    {
        Console.WriteLine("收到 OnClockTick2 的时钟事件!");
    }
}
```

在上述程序中, 对 `Timer` 事件添加了一个新的处理方法。当 `Timer` 事件发生后, 不仅将调用 `OnClockTick()` 方法, 还将调用 `OnClockTick2()` 方法。程序的输出结果如下:

```
收到时钟事件!
收到 OnClockTick2 的时钟事件!
收到时钟事件!
收到 OnClockTick2 的时钟事件!
收到时钟事件!
收到 OnClockTick2 的时钟事件!
收到时钟事件!
收到 OnClockTick2 的时钟事件!
收到时钟事件!
收到 OnClockTick2 的时钟事件!
```

11.2.4 利用事件传递数据

利用事件的产生和处理, 不仅可以激活特定的处理过程, 还可以给这些方法传递一些附加的数据信息。为了达到这一目的, 需要完成以下 3 个步骤:

- (1) 定义一个 `System.EventArgs` 类的子类。
- (2) 在该类中封装需要传递给方法的数据, 可以定义为类的成员变量, 也可以定义为类的属性。

(3) 在事件的触发类中创建该类的实例, 并通过事件将该实例传递给相应的方法。

现在对 11.2.2 节中的时钟事件进行修改, 使产生的时钟事件中包含该事件的计数信息。

因此,需要定义一个新的 ClockTimerArgs 类,该类是对 System.EventArgs 类的继承。具体代码如下:

```
public class ClockTimerArgs : EventArgs
{
    private int tickCount;
    public ClockTimerArgs(int tickCount)
    {
        this.tickCount = tickCount;
    }
    public int TickCount
    {
        get
        {
            return tickCount;
        }
    }
}
```

在 ClockTimerArgs 类中定义了一个名为 tickCount 的私有成员变量,该变量用于记录时钟的计时次数信息。该私有变量可以通过类里定义的一个构造函数进行赋值,并且在事件处理方法中利用 ClockTimerArgs 类的公共访问类型的属性获得该变量的值。


为了能够通过事件传递数据信息,还需要对事件的代理进行修改,示例代码如下:

```
public delegate void TimerEvent(object sender, ClockTimerArgs e);
```

代理所定义的方法标识中的第二个参数的类型由 EventArgs 类改为了 ClockTimerArgs 类,这样事件的产生类就可以向事件处理方法传递该类型的参数信息。同时,还需要对事件的产生类进行如下修改:

```
class ClockTimer
{
    public event TimerEvent Timer;
    public void Start()
    {
        for(int i = 0; i < 5; i++)
        {
            Timer(this, new ClockTimerArgs(i + 1));
            Thread.Sleep(1100);
        }
    }
}
```

在该类中只进行了一处修改,就是将 Timer 事件的第二个参数由 null 改为了 ClockTimerArgs 类型的对象,因此就可以把时钟的计时信息传递给事件的处理方法。

 **说明:** 利用事件的产生和处理,可以给方法传递附加的数据信息。

为了使输出结果体现出这种数据传递,还需要对 OnClockTick()方法进行相应的修改,使其能够打印时钟的计时次数信息,代码如下:


```
public static void OnClockTick(object sender, ClockTimerArgs e)
{
    Console.WriteLine("收到一个时钟计时事件, 这是第{0}个计时事件", e.TickCount);
}
```

通过上述修改之后, 就能够实现利用事件传递数据信息的目的。对 ClockTimer 事件完整的应用程序, 如下所示。

```
//声明使用的命名空间
using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _11._2._4
{
    //定义 Sample 类
    class Sample
    {
        static void Main( )
        {
            ClockTimer clockTimer = new ClockTimer( );
            //向 Timer 中添加方法
            clockTimer.Timer += new TimerEvent(OnClockTick);
            clockTimer.Start( );
        }
        //定义符合类型的 OnClockTick( ) 方法
        public static void OnClockTick(object sender, ClockTimerArgs e)
        {
            Console.WriteLine("收到一个时钟计时事件, 这是第{0}个计时事件",
                e.TickCount);
        }
    }
    //定义继承 EventArgs 类的 ClockTimerArgs 类
    public class ClockTimerArgs : EventArgs
    {
        //私有成员变量
        private int tickCount;
        //定义 ClockTimerArgs 类的构造函数
        public ClockTimerArgs(int tickCount)
        {
            this.tickCount = tickCount;
        }
        //定义 TickCount 属性
        public int TickCount
        {
            get
            {
                return tickCount;
            }
        }
    }
    //定义事件的代理
```

```
public delegate void TimerEvent(object sender, ClockTimerArgs e);
//定义 ClockTimer 类
class ClockTimer
{
    public event TimerEvent Timer;
    public void Start()
    {
        for(int i = 0; i < 5; i++)
        {
            Timer(this, new ClockTimerArgs(i + 1));
            Thread.Sleep(1000);
        }
    }
}
```

当上述程序执行之后，输出结果如下：

```
收到一个时钟计时事件，这是第 1 个计时事件
收到一个时钟计时事件，这是第 2 个计时事件
收到一个时钟计时事件，这是第 3 个计时事件
收到一个时钟计时事件，这是第 4 个计时事件
收到一个时钟计时事件，这是第 5 个计时事件
```

通过以上程序可以看到，每次输出中都包含了时钟的计时次数，达到了传递数据信息的目的。

11.3 本章总结

本章中介绍了代理的概念，并且说明了如何对代理进行定义和使用。在其他的一些资料中，又将代理称为“代表”或者“委托”，但其实表达的意思都是一致的。简单地说，代理就是一种对方法的引用。虽然这种说法从理论上讲并不十分准确，但这样解释有利于读者更快地理解代理的作用。

在使用代理之前，必须定义其特定的方法标识，用来指定代理可以引用的方法类型。代理通常具有以下 3 个属性：

- ❑ 代理所指向方法的类型或者标识；
- ❑ 能够被用于指向某个方法的代理的引用；
- ❑ 代理所指向的实际的方法。

在进行 C#程序设计的时候，正确地运用代理能够提高程序的灵活性和可用性，读者需要仔细学习代理的内容，在实际应用中灵活使用。

本章还介绍了事件的概念，事件为类和类的实例提供了向外界发送通知的能力。通过事件的消息，用户能够把事件和处理事件的代码联系在一起。通过事件，能够让程序正确地处理在执行期间遇到的用户预定义的情况。

事件必须是代理类型的，而且可以将事件声明为多重代理类型。这样就可以保证当一个事件发生的时候可以同时触发多个不同的响应过程，增加了程序设计的便利性。

事件具有许多作用，读者在使用的时候应该不断地积累经验，这样才能设计出更加出

色的程序。

11.4 实战练习

1. 在 Visual Studio 2010 中创建一个控制台应用程序，通过委托方式编写一个进行加减乘除四则运算的程序，当用户输入两个单精度数，然后输入一个运算符号，输出两个单精度数进行运算后的结果。

2. 在 Visual Studio 2010 中创建一个控制台应用程序，使用定时器每隔 10 秒显示一次提示框，显示当前的时间。

第4篇 Windows 程序设计

- ▶▶ 第12章 Windows 应用程序概述
- ▶▶ 第13章 Visual Studio 2010 控件介绍
- ▶▶ 第14章 列表选择控件介绍
- ▶▶ 第15章 数据显示控件
- ▶▶ 第16章 通用对话框
- ▶▶ 第17章 其他常用控件
- ▶▶ 第18章 Windows 应用程序的部署

第 12 章 Windows 应用程序概述

在前面的各章节中，为了便于编程语言的学习，使读者迅速掌握 C#语言的基本要素，列举出的程序大多是控制台程序。但是在实际应用中，所使用的大部分应用程序多为 Windows 应用程序。在后面的章节中，将重点介绍如何将 C#的强大功能与 Visual Studio 相配合，更加迅速、便捷地实现 Windows 应用程序的开发；阐述如何在应用程序中最有效地利用 .NET Framework 类，对 Visual Studio IDE 的用法进行指导，并通过创建实用的示例应用程序来具体说明它们的用法。

12.1 Windows 应用程序

本节将针对 Windows 应用程序的基本概念展开论述，并创建一个 Windows 应用程序。

12.1.1 什么是 Windows 应用程序


从广义范围讲，Windows 操作系统下能运行的应用程序都算是 Windows 应用程序，包括扩展名为 .exe\ .dll 等的应用程序。从狭义范围讲，Windows 应用程序主要是指拥有图形用户界面，并能与用户进行交互的程序。在本章中主要根据狭义范围内的 Windows 应用程序编程进行讨论。

Windows 应用程序可以包含一个窗体或一个父窗体和几个子窗体。只包含一个窗体的界面是单文档界面 (SDI)，而包含一个父窗体和几个子窗体的界面称为多文档界面 (MDI)。使用多个界面的应用程序称为 MDI 应用程序。这种应用程序包含一个父窗体和几个子窗体。

12.1.2 创建 Windows 应用程序

在 Visual Studio 2010 中选择“文件”|“新建项目”命令菜单，出现“新建项目”对话框。选择“Windows 窗体应用程序”作为项目模板，将程序名称换为 Example，然后单击 OK 按钮，一个 Windows 应用程序就创建好了。

图 12.1 显示了在 Visual Studio IDE 的对象浏览器中，项目 Example 的具体组成对象，包括 Form1.Designer.cs、Form1.cs，以及 Program.cs 3 个部分。

 说明：Properties 和 References 文件夹的具体作用属于 C#高级编程范畴，在此处不进行讨论。

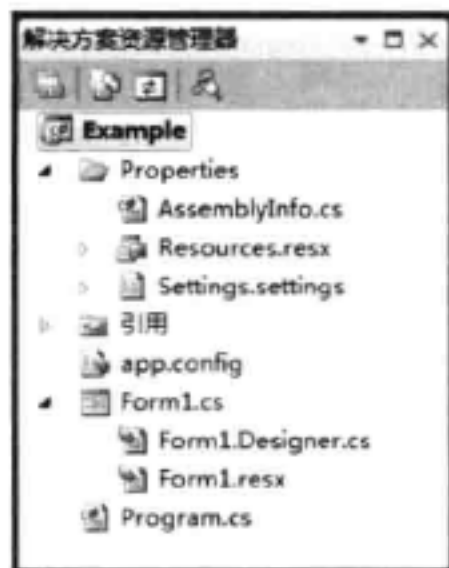



图 12.1 对象浏览器

12.1.3 程序入口文件 Program.cs

Program.cs 是主程序的入口，包括一个 Main() 函数。示例程序如下：

```
static class Program
{
    ///<summary>
    ///应用程序的主入口指针
    ///</summary>
    //单线程单元
    [STAThread]
    static void Main()
    {
        //EnableVisualStyles 通常是 Main() 函数的第一行
        //它的作用是让控件显示出来（包括窗体）
        Application.EnableVisualStyles();
        //作用：在应用程序范围内设置控件显示文本的默认方式（可以设为使用新的 GDI+，还是旧的 GDI）
        //true 使用 GDI+ 方式显示文本，
        //false 使用 GDI 方式显示文本。
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}
```

 注意：[STAThread] 指示应用程序的 COM 线程模型是单线程单元，若为多线程单元则是 [MTAThread]。

12.1.4 窗口程序文件 Form1.cs

自 .NET 2.0 开始 C# 采用了部分类的概念，Form1.Designer.cs、Form1.cs 文件是同一个类的两部分，两个加起来才是一个完整的类。其中，Form1.Designer.cs 是设计器自动生成的，而 Form1.cs 是完成窗口程序的主体部分。在 Form1.cs 中添加 Windows 应用程序的具体功能函数，使 Windows 应用程序符合用户提出的需求。所以，在 Windows 应用程序开发中，一般只修改 Form1.cs 即可，示例程序如下：

```
namespace Example
{
    public partial class Form1 : Form
    {
        //Form1 构造函数
        public Form1()
        {
            //InitializeComponent() 方法一般是 VS.NET 设计器自动生成的 1 个方法
        }
    }
}
```

```

        //完成初始化的任务的方法
        //读者也可以进行手动初始化
        InitializeComponent();
    }
}

```

 注意: InitializeComponent 方法一般是 Visual Studio 设计器自动生成的一个方法。

12.1.5 窗口设计文件 Form1.Designer.cs

Form1.Designer.cs 是设计器自动生成的, 记录 Windows 应用程序中引用的控体和相关属性的设置。一般情况下, 不需要对它进行编辑, 示例程序如下:

```

namespace Example
{
    partial class Form1
    {
        ///<summary>
        ///必需的设计器变量
        ///</summary>
        //定义了一个容器
        //实际编程中大部分窗体的这个变量都一直保持 null
        //仅当添加了组件 (比如 imagelist, 用户自定义的组件), 这个变量才不会是 null。
        private System.ComponentModel.IContainer components = null;

        ///<summary>
        ///清理所有正在使用的资源
        ///</summary>
        ///<param name="disposing">如果应释放托管资源, 为 true; 否则为 false。
        </param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                //如果 components 不为空的话, 释放 components 资源
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        // #region, 在使用 Visual Studio 代码编辑器的大纲显示功能时指定可展开或折叠的代码块
        #region Windows 窗体设计器生成的代码
        ///<summary>
        ///设计器支持所需的方法 - 不要
        ///使用代码编辑器修改此方法的内容
        ///</summary>


```



```

private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    //控件或窗体根据操作系统中字体的大小进行拉伸或缩小
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}
//必须用#endregion 指令终止#region 块
#endregion
}
}

```

 **技巧：** #region、#endregion 组成一对，用于程序的折叠。可增加程序的可读性。在做耗时操作时，为防止白屏，先调用 SuspendLayout() 函数，停止 Form 刷新，操作结束后再调用 ResumeLayout() 函数恢复。

12.2 Windows 应用程序窗体

.NET 框架中提供了一种开发图形用户接口(GUI)的窗口设计体系——Windows Forms (Windows 窗体)。Windows Forms 不同于以往的 WIN32 API 或 MFC，它提供了诸多新的特性，使 Windows 应用程序的开发变得更加方便而简洁，在一定程度上简化了 Windows 程序设计。

12.2.1 C#的 Form 类

C#的窗体是通过 System.Windows.Forms, Form 类或者从 System.Windows.Forms.Form 派生出来的类的对象创建的。窗体能够代表任何类型的窗口，其中包括应用程序主窗口、子窗口，甚至对话框。在代表窗体的 Form 类中封装了创建窗体显示窗体及响应用户窗体所必需的基本功能。

在 12.1 节所创建的项目 Example 中，按 Ctrl+Alt+F5 键编译并运行程序，生成的.exe 文件如图 12.2 所示，只是一个窗口，没有任何功能和特征。如要使这个窗体满足用户需求，并能完成一定的业务逻辑，则需要对这个窗体的属性和事件进行设置。

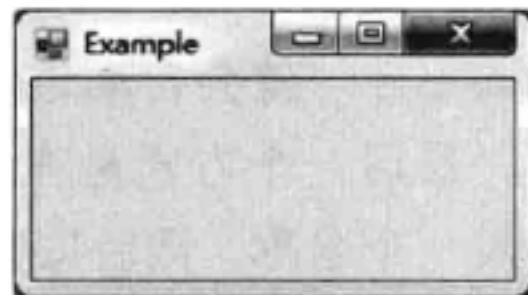


图 12.2 Example.exe

12.2.2 Form 类常用属性

在 Visual Studio IDE 的右侧属性栏中，列出了 Form 类的属性，这些属性用于处理窗体的显示和操作。窗体常用属性如表 12.1 所示。

表 12.1 窗体常用属性

属 性	描 述
AutoScroll	这个属性是个布尔值，用来设置当窗体中的控件内容大于窗体的可见大小时，是否显示滚动条
BackColor	定义窗体的背景颜色
Cursor	当鼠标移动到窗体区域时所显示的光标
Font	定义窗体上文字的字体和字号
Icon	显示在窗体标题栏上的图标
IsMdiContainer	设置应用程序是否为 MDI 应用程序
Text	设定窗体标题
TopMost	设置窗体是否永远在最前方显示

12.2.3 Form 类常用事件

向窗体发送消息时，系统把此消息转化为事件。因此，要处理 Windows 消息，只要使用窗体为该消息注册一个事件处理程序，这样，无论什么时候接收到消息，都会自动调用事件处理程序。窗体的常用事件属性如表 12.2 所示。

表 12.2 窗体常用事件

事 件	描 述
Click	当组件被点击时发生
FormClosed	在窗体关闭后发生
Load	当用户加载组件时发生

处理事件有 3 种基本方式。第 1 种是双击控件，则会进入控件默认事件的处理程序，这个事件对于不同的控件是不同的。如对于窗体（Form）来说，进入的就是 Load 事件。如果控件的默认事件不是需要的事件，则要从属性窗体中的事件列表中选择需要的事件（单击图属性窗体上方的闪电图标按钮，则显示窗体控件的全部事件），在列表中双击该事件，就会自动进入此事件的处理程序，如图 12.3 所示。最后一个方式是在该事件的右侧文本框中，为该事件输入一个名称，然后按 Enter 键，程序就会自动生成一个以输入的名字命名的事件处理程序。

并在 Form1.cs 中添加了如下代码：

```
private void Form1_Load(object sender, EventArgs e)
{
}

```

在 Form1.Designer.cs 中添加了如下代码：

```
this.Load += new System.EventHandler(this.Form1_Load);

```

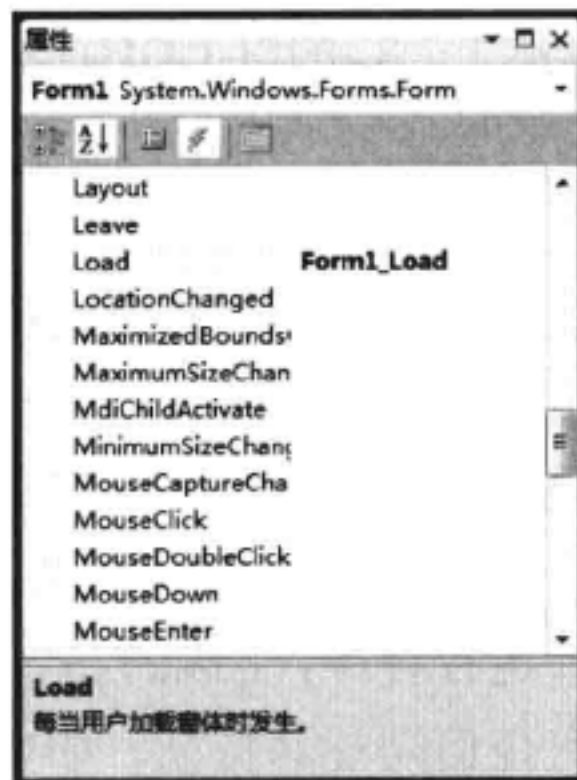



图 12.3 窗体事件处理程序生成

 **说明：**无论使用哪种方式生成的事件处理程序，都可在此程序中填入业务处理逻辑，以满足应用程序的需求。

12.3 为窗体添加控件

当最初创建窗体时，它是空的。要添加功能，则需要向窗体中添加控件，如按钮、文本框等。也就是说，从窗体本身来看，它只是一个可移动的界面而已，其本质就是充当 Windows 各种控件的容器。


12.3.1 Windows 窗体控件

“控件”是窗体上的一个组件，用于显示信息或接受用户输入。利用 .NET 附带的许多控件进行 Windows 应用程序开发，可以使设计用户界面，处理人机交互变得非常简单。.NET 控件大体可以分成以下几类：向用户显示信息的控件、文本编辑控件、文本显示控件、列表选择控件、值设置控件、命令控件、菜单控件、容器控件等。

为了使 Windows 应用程序开发过程简单可见，Visual Studio 提供了大量的控件，这些控件可以按常规功能分为 16 类，具体的分类方法如表 12.3 所示。在后面的章节中，将对常用的控件进行详细的介绍。

表 12.3 Windows 窗体控件

功 能	控 件	功 能	控 件
文本编辑	TextBox 控件	将其他控件分组	GroupBox 控件
	RichTextBox 控件		TabControl 控件
	MaskedTextBox 控件		SplitContainer 控件
文本显示（只读）	Label 控件		TableLayoutPanel 控件
	LinkLabel 控件		FlowLayoutPanel 控件
	StatusStrip 控件	数据显示	DataGridView 控件
	ProgressBar 控件	数据绑定和定位	BindingSource 控件
从列表中选择	CheckedListBox 控件	网页显示	BindingNavigator 控件
	ComboBox 控件		WebBrowser 控件
	DomainUpDown 控件	图形显示	PictureBox 控件
	ListBox 控件	图形存储	ImageList 控件
	ListView 控件	数据的设置	DateTimePicker 控件
	NumericUpDown 控件		MonthCalendar 控件
	TreeView 控件	对话框	ColorDialog 控件
值的设置	CheckBox 控件		FontDialog 控件
	RadioButton 控件		OpenFileDialog 控件
	TrackBar 控件		PrintDialog 控件
命令	Button 控件		PrintPreviewDialog 控件
	NotifyIcon 控件		FolderBrowserDialog 控件
	ToolStrip 控件		SaveFileDialog 控件
菜单控件	MenuStrip 控件	用户帮助	HelpProvider 控件
	ContextMenuStrip 控件		ToolTip 控件
将其他控件分组	Panel 控件	音频	SoundPlayer 控件

说明：在.NET中，大多数控件都派生于 System.Windows.Forms.Control 类。这个类定义了控件的基本功能，因此在控件中有很多属性和事件是相同的。

下面将对控件的这些通用的属性与事件进行具体描述。

12.3.2 控件常见属性

System.Windows.Forms.Control 类是大多数控件的基类，它具有很多属性，其他控件通过继承或重写它的属性，进行定制的操作。如表 12.4 所示，列出了 Control 类最常见的属性。大多数的控件都具有这些属性，在此对这些属性进行统一的说明，在后面对控件逐一介绍时，将不再赘述。

表 12.4 窗体常用属性

属 性	描 述
Anchor	定义了控件的父控件大小发生变化时，控件的大小将发生何种变化
BackColor	定义了控件的背景色
BackgroundImage	定义了控件的背景图像
Bottom	控件的下边缘与父控件的上边缘间的距离
Dock	定义了控件停靠到父控件的哪一个边缘
Enable	定义了控件是否可被禁用
Left	定义了控件的左边缘的 x 轴坐标
Location	定义了控件的坐标
Name	定义了控件的名称
Size	定义了控件的高度和宽度
TabIndex	定义了控件的 Tab 键顺序
Text	定义了此控件的相关文本
Visible	指示该控件是否可见

12.3.3 控件常用公共事件

事件是可以通过代码响应或“处理”的操作，通常情况下，事件可由用户操作（如单击鼠标或按某个键）、程序代码或系统生成。每个窗体和控件都公开了一组预定义事件，可以根据这些事件进行编程。

由控件引发的事件通常是用户的操作引起的，例如，当用户单击某个控件时，该控件就会生成一个事件，说明用户执行了一个什么操作，而程序员在事件处理程序中添加的内容就为该控件提供了一个功能。

Control 类中定义了大多数控件的一些比较常见的事件，如表 12.5 列出了常见的事件。

表 12.5 窗体常见事件

事 件	描 述	事 件	描 述
Click	单击控件时发生	Leave	焦点离开控件时发生
DoubleClick	双击控件时发生	Paint	重绘控件时发生
DragDrop	完成拖放操作时发生	TextChanged	Text 属性值更改时发生

续表

事 件	描 述	事 件	描 述
DragEnter	将对象拖入控件的边界时发生	Validated	在控件完成验证时发生
DragLeave	将对象拖出控件的边界时发生	Validating	在控件正在验证时发生
Enter	进入控件时发生	VisibleChanged	Visible 属性值更改时发生
GetFocus	控件接收焦点时发生		

12.3.4 向窗体添加控件的方法

窗体的设计是通过将控件添加到窗体表面来定义用户界面的。

向窗体添加控件，共有 3 种方法。第 1 种方法是，在 Visual Studio IDE 左边的工具栏里选择想要添加的控件后，将此控件拖曳到窗体中，选择合适位置释放。第 2 种方法是，在工具箱中双击想要添加的控件，则此控件会出现在窗体的默认位置（如左上角）。第 3 种方法是，在工具箱单击想要添加的控件，然后在窗体上的适当位置拖曳鼠标，选择控件的大小后，松开鼠标，则控件添加到窗体上。这种方法的好处是，可以在添加控件时，设置控件的大小。

12.3.5 调整控件的布局

将控件添加到窗体后，需要对窗体进行布局，使窗体美观。拖动窗体上的控件，可以看到标记线会自动标记出控件的对齐方式，如图 12.4 所示。也可以通过菜单栏上的布局按钮，如图 12.5 所示，对控件进行布局。



图 12.4 控件标记线

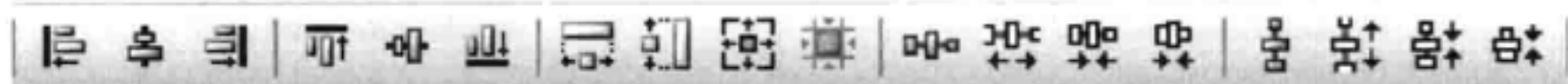


图 12.5 控件布局按钮

这些布局按钮从左至右的功能依次为：控件左对齐、控件中心垂直对齐、控件右对齐、控件上对齐、控件水平中心对齐、控件下对齐、控件等长、控件等宽、控件大小相等、控件按标志线决定大小、控件水平距离相等、控件水平距离增大、控件水平距离缩小、取消控件间水平距离、控件垂直距离相等、控件垂直距离增大、控件垂直距离缩小、取消控件间垂直距离。

对控件布局后，控件被设置在了窗体相应的位置上，但是在应用程序运行时，窗体的大小却不可能永远和设定的大小相同，用户可以自行任意调节窗体的大小，当窗体的大小

被改变时窗体上控件的布局不再如程序初始化时那样工整，而是发生了空间错落排序的状态，如图 12.6、图 12.7 和图 12.8 所示。



图 12.6 应用程序初始化时的窗体



图 12.7 窗体拉伸后

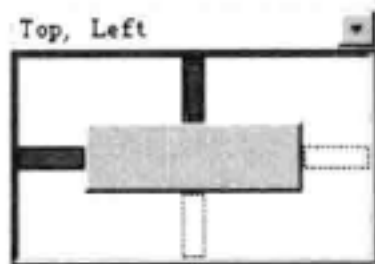
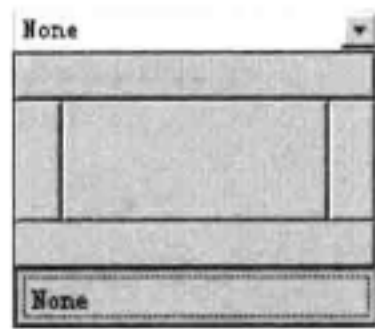


图 12.8 窗体缩小后

说明：为了避免上述现象的发生，需要对窗体的相关属性进行设置。与窗体布局相关的属性主要有 `Anchor` 和 `Dock` 两个。

`Anchor` 属性是用来设置控件绑定到的容器的边缘，并确定控件如何随其父级一起调整大小。在属性窗口单击 `Anchor` 右边的下三角按钮，有一个非常形象的窗口可以让开发人员选择 `Top`、`Bottom`、`Left` 和 `Right` 4 个属性值的组合。这 4 个属性的作用是当读者选择了表示控件一个方向的属性，那么控件的这个方向的边与窗口同方向的边的距离将会被锁定，例如当选择为 `Left` 时，控件的 `Left` 边与窗口的 `Left` 边的距离将不会随窗口大小的改变而改变。这样锁定的结果就是，如果对称的一对边（`Left` 和 `Right`、`Top` 和 `Bottom`）中锁定了其中一个，当窗口大小改变时，控件会改变自身的位置，确保与窗口对应边的距离不变。如果将对称的一对边都进行了锁定，当窗口大小改变时，控件对应方向的大小会随之改变，即被拉伸或压缩。对 `Anchor` 属性值的设定如图 12.9 所示，默认值为 `Top` 和 `Left`。

使用 `Dock` 属性可以设置控件停靠到父容器的哪一个边缘。对 `Dock` 属性值的设定如图 12.10 所示，默认值为 `None`。一个控件可以停靠到其父容器的一个边缘或者停靠到所有边缘并充满父容器。例如，如果将该属性设置为 `DockStyle.Left`，控件的左边缘将停靠到其父控件的左边缘。

图 12.9 `Anchor` 属性值的设定图 12.10 `Dock` 属性值的设定

因此，在开发 Windows 应用程序时，经常会使用 `Dock` 属性来设置控件的靠边方式。因为它使用起来非常方便，能帮助开发人员自动定位，不管窗口形状发生任何变化，所控制的控件也会自动进行相应的调整。但是，当在同一个 `Form` 中给多个控件设置 `Dock` 属性时，一个问题就出现了。

例 1：`Form` 中有两个 `Panel` 控件，`Panel1` 和 `Panel2`。将 `Panel1` 的 `Dock` 属性设为 `Left`，把 `Panel2` 的 `Dock` 属性设为 `Fill`。如希望 `Panel1` 能靠左边框显示，`Panel2` 则填满剩下的区域。但是程序运行时发现 `Panel2` 填满了整个 `Form`，而 `Panel1` 则正常靠左显示，却把部分 `Panel2` 的内容给覆盖了。

例 2: Form 中有一个 Panel (Dock 属性为 Left) 和一个 StatusStrip (Dock 属性为 Bottom) 控件。一般情况下希望 StatusStrip 能紧靠底部, 完整显示在 Form 中, 但运行结果往往是 StatusStrip 只显示在 Panel 以外的区域中。

出现以上两种情况是因为, 在 Visual Studio 中控件依照 Z 顺序停靠。也就是说, 离 Form 层次越近的控件, 停靠的优先级越高。层越高的控件会在下面层次停靠后所剩下的空间中再进行停靠动作。以上两个例子的解决办法也就变得很简单了: 在界面设计时, 选中例 1 中的 Panel1 和例 2 中的 StatusStrip 两个控件, 然后再在右键菜单中进行“置于底层”的操作, 一切问题都迎刃而解了。

还有一点值得注意的就是, 在 Visual Studio 中 Anchor 和 Dock 属性是互相排斥的。每次只可以设置一个属性, 最后设置的属性优先。

12.4 Windows 应用程序编程示例

在这一节中, 将创建一个简单的 Windows 应用程序, 帮助读者理解控件的属性及事件。在这个应用程序中, 将在窗口上设置一个按钮控件, 单击该控件时, 弹出一个对话框, 上面写着“欢迎进入 Windows 应用程序编程的世界”。

12.4.1 创建一个窗体

关于如何创建一个窗体, 在本章的第一节, 已经对具体步骤和相应的代码段进行了具体的介绍, 这里将不再赘述。读者可参考在本章第一节中创建的项目“12.1”。

12.4.2 用属性控制窗体外观

调整窗体大小, 使窗体大小满足应用程序的要求。调整窗体的大小主要有两种方法。第一种方法是, 通过调整窗体的 Size 属性来调整窗体的大小。通过为 Size 属性设置新的属性值, Windows 窗体的大小将随之改变。这种方法也可以在编程中应用, 在应用程序运行过程中, 以编程方式改变窗体的大小。另一种方法是, 在 Windows 窗体设计器中改变窗体的大小。首先在 Windows 窗体设计器中, 单击该窗体以选定它, 单击并拖动窗体边框上出现的 8 个尺寸柄中的一个。尺寸柄看起来像个小白框, 当鼠标指向它时, 会变成小箭头。

下面, 通过修改窗体的具体属性来修改窗体的外观, 使窗体更具有友好性, 具体修改如表 12.6 所示。

表 12.6 窗体属性设置

属 性	值	属 性	值
BackColor	ControlLight	Text	Example12.1
StartPosition	CenterScreen		

12.4.3 向窗体添加控件

在 Visual Studio IDE 左边的工具栏里，显示出了 .Net 提供的常用控件。选择 Button 控件拖曳到窗体中，选择合适位置释放鼠标。Button 控件是 Windows 应用程序编程中使用最频繁的控件，主要用于触发事件。对于 Button 控件的详细介绍，将在 13 章中讲解。对 Button 控件的属性进行设置，改变 Button 的外观，使其更加具有用户界面友好性，具体细节如表 12.7 所示。

表 12.7 Button控件属性设置

属 性	值	属 性	值
Name	btnExample	Text	显示信息
Anchor	Botton, Right		

12.4.4 添加控件事件处理程序

事件处理程序是代码中的过程，用于确定在事件发生时程序要执行的操作。当引发一个事件时，程序会自动执行该事件的处理程序。

在窗体设计器中，双击窗体上的 Button 控件，则.NET 为 Button 自动生成了 click 事件，在事件中添加处理程序，当单击 Button 控件时，会自动运行处理程序中编辑的功能。具体代码如下：

```
//第一个参数 sender 提供对引发事件的对象的引用
//第二个参数 e 传递特定于要处理的事件的对象
private void btnExample_Click(object sender, EventArgs e)
{
    //MessageBox.Show() 的功能是显示一个消息对话框
    MessageBox.Show("欢迎进入 Windows 应用程序编程的世界。");
}
```

12.4.5 查看窗体运行效果

单击菜单栏上的“启动调试”按钮，程序编译成功后运行结果如图 12.11 所示。单击 Button 控件，即图中的“显示信息”按钮，运行结果如图 12.12 所示。至此，完成了一个简单的 Windows 应用程序的开发。



图 12.11 Windows 应用程序窗体



图 12.12 Windows 应用程序对话框


12.5 本章总结

在本章中,脱离了单纯的 C#语法的学习,开始与实际应用相结合,进行常见的 Windows 应用程序的开发。本章主要讲解了如何利用 Visual Studio.Net 平台进行快速的 Windows 应用程序开发。同时对 Windows 应用程序中的窗体类和控件的概念进行了简要的概述。

12.6 实战练习

1. 在 Visual Studio 2010 中新建一个“Windows 窗体应用程序”,在默认添加的 Form1 窗体中添加一个按钮,设置按钮显示文字为“登录”,为“登录”按钮编写程序,显示一个欢迎使用的提示信息。

2. 在上题的 Form1 窗体中再添加一个显示文字为“退出”的按钮,为该按钮编写程序,关闭应用程序。

提示: 使用 Application 类的静态方法 Exit()可退出应用程序。

第 13 章 Visual Studio 2010 控件介绍

利用 .NET 附带的许多控件进行 Windows 应用程序开发，可以使设计用户界面、处理人机交互变得非常简单。 .NET 控件大体可以分成几类：向用户显示信息的控件、文本编辑控件、文本显示控件、列表选择控件、值设置控件、命令控件、菜单控件和容器控件等。

13.1 接收输入的文本编辑控件

在 Visual Studio 中，文本编辑控件主要包括显示设计时输入的 TextBox 控件，使文本能够以 RTF 格式显示的 RichTextBox 控件和约束用户输入格式的 MaskedTextBox 控件。由于 MaskedTextBox 控件在 Windows 应用程序开发中并不常被用到，所以在本节中主要介绍 TextBox 控件和 RichTextBox 控件。

13.1.1 TextBox 控件的作用

当 Windows 应用程序不要求用户修改相应的信息时，可使用 Label 控件显示它。当 Windows 应用程序允许用户修改相应信息时，可使用 TextBox 控件显示相应信息。 TextBox 控件可以用来获取用户的输入信息或者显示文本信息。 TextBox 控件通常与 Label 控件搭配使用，使用 Label 控件标注 TextBox 控件的内容。

TextBox 控件的用法非常灵活，它既可以用来编辑文本信息，也可以通过对其相应属性进行设置，使其成为只读控件。在 TextBox 控件中既可以显示简单的单行的短文本，也可以显示复杂的多行长文本，并且可以通过对 TextBox 控件的相应属性进行设置，使文本可换行，既符合控件的大小又能添加基本的格式设置。

TextBox 控件在 Visual Studio 中的命名空间为 System.Windows.Forms。它在 Visual Studio 工具箱中的图示如图 13.1 所示，将 TextBox 控件拖曳到窗体选定后，如图 13.2 所示。下一步，可在 Visual Studio 的属性窗体中对 TextBox 的属性进行设置，使其达到应用程序的具体需求。



图 13.1 TextBox 控件

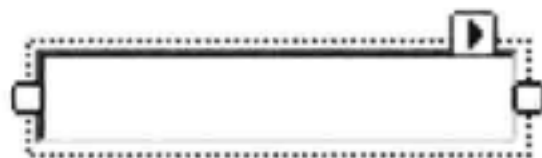



图 13.2 TextBox 控件

13.1.2 怎样获取或设置 TextBox 控件的内容

TextBox 控件中最重要的属性是 Text 属性，该属性返回一个 string 类型的对象，代表显示在 TextBox 控件中的文本内容。

说明：用户能输入的最大字符数为 2147483646（2GB）或一个基于可用内存的数目，两数之间选取较小者。

Text 属性可以在设计时使用“属性”窗口设置，在运行时用代码设置，或者在运行时通过用户输入来设置。也可以在运行时通过读取 Text 属性来检索文本框的当前内容。具体实现方法如下所示。

❑ 读取文本框内容：

```
private string getText()
{
    //获取文本框中内容
    string textContent="";
    textContent=this.textBox1.text;
    return textContent
}
```

❑ 设置文本框内容：

```
private void setText(string textContent)
{
    //设置文本框中内容
    this.textBox1.tex = textContent ;
}
```

AppendText()方法是在文本框控件中的常用方法，可以使用此方法向当前文本框文本的结尾添加文本，具体使用方法如下所示。

```
private void AppendTextBox1text(string apppendedText)
{
    //向文本框中添加内容
    this.textBox1.AppendText (apppendedText);
}
```

AcceptsReturn 属性是用来指示在 TextBox 控件中按 Enter 键时，TextBox 控件所产生的反应。当 AcceptsReturn 属性为 true 时，TextBox 控件中将创建一行新的文本。当 AcceptsReturn 属性为 false 时，则 TextBox 控件将被激活，可以在属性窗体里对此属性进行设置，也可以在代码中进行设置，具体的设置代码如下所示。

```
private void setAcceptsReturn()
{
    //使文本框接收回车键
    this.textBox1. AcceptsReturn= false;
}
```

AcceptsTab 属性是用来指示在 TextBox 控件中按 Tab 键时, TextBox 控件所产生的反应。当 AcceptsTab 属性为 true 时, 此时 TextBox 控件中将键入一个 Tab 字符。当 AcceptsTab 属性为 false 时, 则此时窗体中的控件将按照 TabIndex 属性所规定的顺序被激活 (即激活窗体中的下一个控件), 可以在属性窗体里对此属性进行设置, 也可以在代码中进行设置, 具体的设置代码如下所示。


```
private void setAcceptsTab()
{
    //使文本框可接受 Tab 键
    this.textBox1.AcceptsTab= false;
}
```

13.1.3 TextBox 控件也可输入多行文本

TextBox 控件中如何实现多行文本的输入, 也是一个一直困扰初学者的问题。如果需要在 TextBox 控件中输入多行文本, 则首先需要将 Multiline 属性置为 true。可以使用下面的代码, 或在属性窗体中进行设置。

```
private void setMultiline ()
{
    //设置文本框不能多行输入
    this.textBox1. Multiline= false;
}
```

当 Multiline 属性为 true 时, TextBox 控件中可以输入多行文本。当 Multiline 属性为 false 时, TextBox 控件中则不能输入多行文本。

 **技巧:** 多行文本的内容不能在 Text 属性中设置, 而应该在 Lines 属性中再设置。

单击 Lines 属性旁边的向下箭头, 则会打开相应的“字符串集合编辑器”对话框, 如图 13.3 所示。在其中编写相应的多行文本。集合编辑器里的每一项就是多行文本框中的一行。

也可以使用代码对文本框中的内容进行添加, 具体如下面的代码所示。

```
private void setMultiline ()
{
    //当文本框可接收多行数据时, 设置多行数据内容
    this.textBox1.Text = "第一行文本\r\n 第二行文本\r\n 第三行文本";
}
```

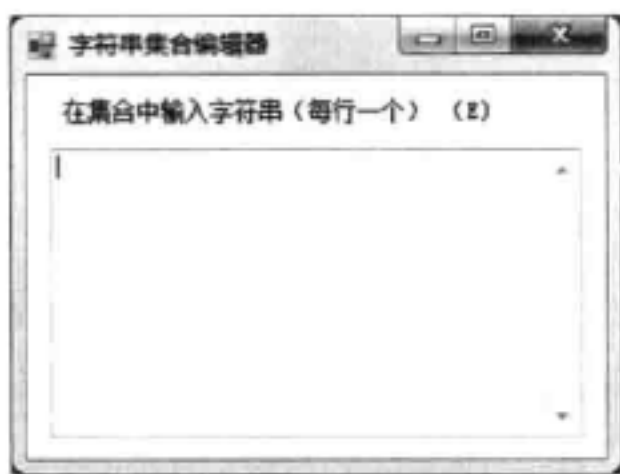



图 13.3 “字符串集合编辑器”对话框

运行效果图如图 13.4 所示。

 **注意:** 将 Multiline 属性设置为 true 是非常重要的, 如果 Multiline 属性为 false 时, 则运行效果如图 13.5 所示。

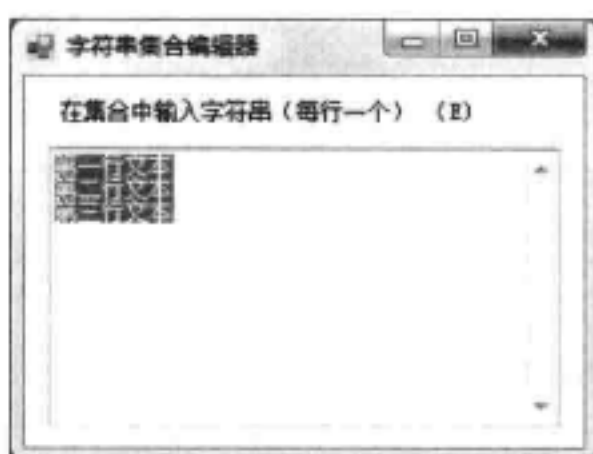


图 13.4 多行文本 TextBox 控件

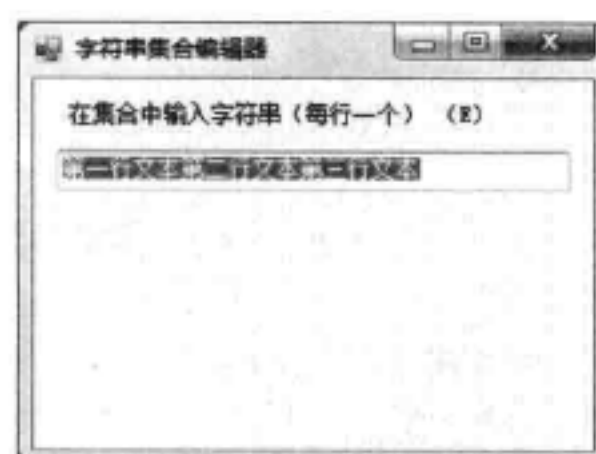


图 13.5 单行文本 TextBox 控件

ScrollBars 属性是一个 ScrollBars 类型的枚举值，它指示了在 TextBox 控件中将出现和使用滚动条，ScrollBars 的具体成员列表如表 13.1 所示。

表 13.1 ScrollBars 成员列表

成 员	说 明	成 员	说 明
Both	显示垂直滚动条和水平滚动条	Horizontal	显示垂直滚动条
None	不显示滚动条	Vertical	显示水平滚动条

在下面的代码中，以编程的方式设置了 TextBox 控件的滚动条。

```
private void setMultiline ()
{
    //使文本框可以显示水平滚动条和垂直滚动条
    this.textBox1.ScrollBars = ScrollBars.Both;
}
```

当将文本框设置为多行文本框时，无论将 ScrollBars 的属性设置为何值，TextBox 控件中均不会出现水平滚动条。

WordWrap 属性表示的是当 TextBox 控件的 Multiline 属性为 true 时，TextBox 控件文本是否可以自动换行。当 WordWrap 属性为 true 时，TextBox 控件中的文本可以自动换行。当 WordWrap 属性为 false 时，TextBox 控件中的文本不可以自动换行。

技巧：当将 WordWrap 属性设置为 true 时，ScrollBars 的属性最好不要设置为 Vertical 或 Both。否则，WordWrap 属性设置将没有意义。

13.1.4 选择 TextBox 控件中的文本

SelectionLength 属性代表的是当前文本框中选定的字符数，其最大值等于 TextBox 控件中的字符个数。而 SelectionStart 属性代表的是当前文本框中选定的文本起始点的索引值，其最小值是 0，最大值是 TextBox 控件中的字符个数。通过使用这两个属性，可以实现 TextBox 控件的很多功能，比如控制 TextBox 控件中的插入点，或者在 TextBox 控件中选择文本。

在默认状态下，TextBox 控件在启动时，文本框内的默认插入点在任何现有文本的左侧，控件内的文本是全部选定的。如果想控制 TextBox 中选择的文本，可以通过编程方式选择文本。具体代码如下所示。


```
private string setSelectText()
{
    //将 SelectionStart 属性设置为要选择的文本的开始位置
    //SelectionStart 属性是一个数字，它指示在文本字符串内的插入点，值为 0 表示最左边的位置
    //如果将 SelectionStart 属性设置为等于或大于文本框内的字符数，则插入点放在最后一个字符之后
    textBox1.SelectionStart = 0;
    //将 SelectionLength 属性设置为要选择的文本的长度
    //SelectionLength 属性是一个设置插入点宽度的数值
    //如果将 SelectionLength 设置为大于 0 的数，则会从当前插入点处开始选择该数目的字符
    textBox1.SelectionLength = textBox1.Text.Length
    //通过 SelectedText 属性访问选定的文本
    return textBox1.SelectedText;
}
```

当文本框失去焦点而后又再次获得焦点时，插入点为用户上一次放置的位置。而在实际应用中，用户通常希望当文本框再次获得焦点时，插入点为现有文本的后面。同样地，也可以通过设置 SelectionStart 和 SelectionLength 属性来满足用户的需求。当 SelectionStart 属性为零时，则插入点紧挨第一个字符的左边。

```
private void setInsertedPoint ()
{
    //设置文本框中字符串选择的起始位置
    textBox1.SelectionStart = 0;
    //设置文本框中字符串选择的长度
    textBox1.SelectionLength = 0;
}
```

在通常情况下，文本框中最多可输入 2147483646（2GB）个字符。但是在大部分应用程序中，并不希望用户可以向文本框中输入太多的字符，这时，可以通过设置 TextBox 控件的 MaxLength 属性来设置可以输入到 TextBox 控件中的文本数量。

TextLength 属性返回了一个 int 值，它代表了 TextBox 控件中文本的长度。


 注意：等式 `this.textBox1.Text.Length == this.textBox1.TextLength` 是恒成立的。

13.1.5 设置 Textbox 控件为密码框

在 Windows 应用程序的开发中，通常需要用到密码输入的功能。在 Windows 窗体开发中，密码框也是一种文本框。将文本框设置为密码框时，需要对文本框的 PasswordChar 属性进行设置。PasswordChar 属性返回一个字符型的对象，此对象代表了屏蔽 TextBox 控件中的文本的密码字符。通过使用 PasswordChar 属性可帮助确保实现以下功能：用户输入密码时如有其他人在观看，他们将无法知道输入的密码。

首先，将 TextBox 控件的 PasswordChar 属性设置为某个特定字符，PasswordChar 属性指定在文本框中显示的字符。例如，如果希望在密码框中显示星号，请在“属性”窗口中将 PasswordChar 属性指定为“*”。无论用户在文本框中输入什么字符，都显示为星号。

通过设置 `MaxLength` 属性可确定在文本框中输入多少字符。如果超过了最大长度，系统会发出声响，且文本框不再接受任何字符。

 **技巧：**通常情况下，不需要设置此属性，因为黑客会利用密码的最大长度来试图猜测密码。

下面的代码示例演示了如何初始化一个文本框，此文本框可接受最长 6 个字符的字符串，并显示星号来替代该字符串。

```
//初始化文本框
private void InitializeMyControl()
{
    textBox1.Text = "";
    //将文本框设置为密码框
    textBox1.PasswordChar = '*';
    textBox1.MaxLength = 6;
}
```

13.1.6 检查 TextBox 控件的输入值

`CausesValidation` 属性是一个 `boolean` 类型的值，它表示在 `TextBox` 控件中是否需要执行验证。当 `TextBox` 控件被激活时需要执行验证，`CausesValidation` 属性值为 `true`。当 `TextBox` 控件被激活时不需要执行验证，`CausesValidation` 属性值为 `false`。可以在属性窗体里对此属性进行设置，也可以在代码中进行设置，具体的设置代码如下所示。

```
private void setCausesValidation(bool causesValidation)
{
    this.textBox1.CausesValidation= causesValidation;
}
```

当 `CausesValidation` 属性值为 `true` 时，`TextBox` 控件获得焦点时将触发 `Validating` 事件和 `Validated` 事件，`Validating` 事件是在控件验证时发生的事件，而且在控件验证后 `Validating` 事件比 `Validated` 事件提前被触发。关于这两个事件的定义如下所示。

```
private void textBox1_Validated(object sender, EventArgs e)
{
    ...//在控件验证时发生的事件
}
private void textBox1_Validating(object sender, CancelEventArgs e)
{
    ...//在控件验证后发生的事件
}
```

13.1.7 支持格式化的 RichTextBox 控件

在上文中讲到的 `TextBox` 控件为在该控件中显示的或输入的文本提供一种格式化样式。若要显示多种类型的带格式文本，则需要使用功能更强的 `RichTextBox` 控件。

RichTextBox 控件在 .NET 中命名空间为 System.Windows.Forms。RichTextBox 控件在 Visual Studio 的工具箱中的图示如图 13.6 所示，将 RichTextBox 控件拖曳到窗体选定后，如图 13.7 所示。然后，可在 Visual Studio 的属性窗体中对 RichTextBox 的属性进行设置，使其达到应用程序的具体需求。

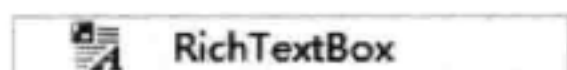


图 13.6 RichTextBox 控件

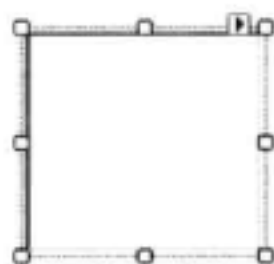


图 13.7 RichTextBox 控件

13.1.8 选择 RichTextBox 控件中的文本

SelectedText 属性是一个 string 类型的对象，它表示在当前 RichTextBox 控件中所选择的字符串。SelectionFont 属性表示的是当前 RichTextBox 控件中所选择的字符串的字体，SelectionColor 属性表示的是当前 RichTextBox 控件中所选择的字符串的颜色，以上 3 个属性都可以通过代码进行获取和设置，具体代码如下所示。

```
private void SetSelectedTextProperty()
{
    //设置选择字符串的颜色
    this.richTextBox1.SelectionColor= Color.Red;
    //设置选择字符串的字体
    this.richTextBox1.SelectionFont= new Font("Arial", 16);
    this.textBox1.Text =this.richTextBox1.SelectedText
}
```

RichTextBox 控件的 Text 属性返回的是 RichTextBox 控件中的文本内容，可是 RichTextBox 控件显示的文本是 Rtf 格式的文本文件。有的时候，用户可能不仅仅希望只获取控件中的文本，还希望获得包含所有 RTF 格式的代码，这时就可以使用 RichTextBox 控件的 Rtf 属性。关于 Rtf 属性的获取和设置不能在属性窗体中进行，可以使用如下的代码获取 RichTextBox 控件的 Rtf 属性。

```
private string getRtf()
{
    string rtfText="";
    //获取 RichTextBox 控件中文本的 rtf 格式
    rtfText=this.richTextBox1.Rtf;
    return rtfText
}
```

SelectedRtf 属性是一个 string 类型的对象，它表示在当前 RichTextBox 控件中所选择的 RTF 格式的字符串。对于 SelectedRtf 属性的获取和设置也只能通过代码实现，具体代码如下所示。


```
private string getSelectedRtf()
{
```



```

string selectedRtfText="";
//获取 RichTextBox 控件中选定文本的 rtf 格式
selectedRtfText=this.textBox1.SelectedRtf;
return selectedRtfText
}

```

说明: SelectedRtf 属性与 Rtf 属性和 SelectedText 属性有着很大的相关性,可以说是 Rtf 属性和 SelectedText 属性相结合。

AutoWordSelection 属性是用来指示在 RichTextBox 控件中选择文本时, RichTextBox 控件所产生的反应。当 AutoWordSelection 属性为 true 时,将启动自动选择字词,此时在 RichTextBox 控件中选择文本的任何部分,都将导致选择整个单词。可以在属性窗体里对此属性进行设置,也可以在代码中进行设置,具体的设置代码如下所示。

```

private void setAutoWordSelection ()
{
    this.richTextBox1.AutoWordSelection= false;
}

```


13.1.9 在 RichTextBox 控件中撤销上次操作

CanUndo 属性是用来指示在 RichTextBox 控件中是否可以撤销前一操作。CanUndo 属性是一个 boolean 类型的变量,该属性只可以在代码中进行设置,具体的设置代码如下所示。

```

private void setCanUndo ()
{
    //设置 RichTextBox 控件允许撤销操作
    this.richTextBox1.CanUndo= true;
}

```

说明:当 CanUndo 属性为 true 时,则程序可以调用 Undo()方法来撤销应用在 RichTextBox 控件中的上一个操作。

在下面的代码中,单击窗体里的一个按钮,将会执行 RichTextBox 控件的 Undo 操作,具体代码如下所示。

```

private void button1_Click(object sender, EventArgs e)
{
    //判断该文本框是否可撤销操作
    if(this.richTextBox1.CanUndo == true)
    {
        //撤销最后的操作
        this.richTextBox1.Undo();
        //从该文本框的撤销缓冲区中清除关于最近操作的信息
        this.richTextBox1.ClearUndo();
    }
}

```

CanRedo 属性用来指示在 RichTextBox 控件中是否可以重新调用应用在该控件中的被

撤销的前一操作。CanRedo 属性是一个 boolean 类型的变量，当 CanRedo 属性为 true 时，则程序可以调用 Redo 方法来重新调用应用在 RichTextBox 控件中的被撤销的前一操作。CanRedo 属性只可以在代码中进行设置，具体的设置代码如下所示。

```
private void setCanRedo ()
{
    this.richTextBox1.CanRedo= true;
}
```

RedoActionName 属性是一个字符串类型的对象，用来指示在 RichTextBox 控件中调用 Redo 方法后，重新在控件中应用的操作名字。操作名字主要有“增”、“删”、“改”3 大类型。


在下面的代码中，单击 button1 按钮，将会执行 RichTextBox 控件的撤销操作。单击 button2 按钮，将会执行 RichTextBox 控件的上次撤销的操作，并在一个 TextBox 中显示操作的名称。如果名称为“”时，表示没有可执行的操作。

```
private void button1_Click(object sender, EventArgs e)
{
    //判断该文本框是否可撤销操作
    if(this.richTextBox1.CanUndo == true)
    {
        //撤销最后的操作
        this.richTextBox1.Undo();
    }
}

private void button2_Click(object sender, EventArgs e)
{
    //判断该文本框是否可执行 Redo 操作
    if(richTextBox1.CanRedo == true)
    {
        //执行 Redo 操作
        this.richTextBox1.Redo();
        //在 textBox1 中显示操作的名称
        this.textBox1.text=richTextBox1.RedoActionName;
    }
}
```

13.1.10 拖放 RichTextBox 控件中的文本

AllowDrop 属性是一个 boolean 类型的变量，它表示在 RichTextBox 控件中是否允许执行拖放操作，当允许执行拖放操作时，AllowDrop 属性值为 true。

 **技巧：**当 AllowDrop 属性为 true 时，则可以通过处理 RichTextBox 控件的 DragEnter 事件和 DragDrop 事件来完成相应的拖放操作。

DragEnter 事件是在将对象拖放到控件时发生，在 DragEnter 事件中添加处理程序，对于所拖曳的文本文件实行 Move 操作，具体代码如下所示。

```
//鼠标进入另一个控件时引发 DragEnter 事件
```



```
private void richTextBox1_DragEnter(object sender, System.Windows.Forms.
DragEventArgs e)
{
    //确保所拖动的数据类型为可接受的类型
    if (e.Data.GetDataPresent(DataFormats.Text))
    {
        //执行拖放操作的最后结果是移动
        e.Effect = DragDropEffects.Move;
    }
    else
    {
        //执行拖放操作的最后结果是无反应
        e.Effect = DragDropEffects.None;
    }
}
```

DragDrop 事件是在完成拖放时发生，在下面的代码中，可将前面拖动的文本复制到 RichTextBox 控件的插入点处。

```
//完成拖放操作时引发 DragDrop 事件
private void richTextBox1_DragDrop(object sender, System.Windows.Forms.
DragEventArgs e)
{
    int i;
    String s;
    //获取文本框中选择文字的起始字符序号
    i = richTextBox1.SelectionStart;
    //截取文本框中所选文字部分及以后的部分
    s = richTextBox1.Text.Substring(i);
    //截取文本框中所选文字部分的前面部分，设置给文本框中的文本
    richTextBox1.Text = richTextBox1.Text.Substring(0,i);
    //向文本框中添加拖曳的文本
    richTextBox1.Text = richTextBox1.Text + e.Data.ToString();
    //向文本框中追加所选文本的后面的文本
    richTextBox1.Text = richTextBox1.Text + s;
}
```

13.1.11 设置 RichTextBox 控件中的文本格式

SelectionBullet 属性是一个 boolean 类型的变量，它表示是否对 RichTextBox 控件中当前选定的文本内容应用项目符号样式，当 SelectionBullet 属性为 true 时，则在 RichTextBox 控件中当前选定的文本内容中是项目符号样式。BulletIndent 属性是一个 int 类型的对象，它代表着在对 RichTextBox 控件中的文本应用项目符号样式时，在项目符号后面插入的像素值。

在下例中，将在 RichTextBox 控件中输入一些文本，对于其中的一些文本指定了字体、颜色和格式。具体代码如下所示。

```
private void button1_Click(object sender, EventArgs e)
{
```

```

//清除 RichTextBox 控件中的内容
richTextBox1.Clear();
//设置 RichTextBox 控件中的字体的大小和样式
richTextBox1.SelectionFont = new Font("Arial", 16);
//向其中添加文本
richTextBox1.SelectedText = "列举三原色, 并以相应的颜色打印出来" + "\n";
richTextBox1.BulletIndent = 30;
//设置 RichTextBox 控件中的第一项的字体的大小和样式
richTextBox1.SelectionFont = new Font("Arial", 12);
//将 SelectionBullet 属性置为 true, 则下面的文本将应用项目符号样式
richTextBox1.SelectionBullet = true;
//设置 RichTextBox 控件中的第一项的内容的颜色
richTextBox1.SelectionColor = Color.Red;
//设置 RichTextBox 控件中的第一项的内容
richTextBox1.SelectedText = "红" + "\n";
//对以下各项也执行相同的操作
richTextBox1.SelectionFont = new Font("Arial", 12);
richTextBox1.SelectionColor = Color.Yellow;
richTextBox1.SelectedText = "黄" + "\n";
richTextBox1.SelectionFont = new Font("Arial", 12);
richTextBox1.SelectionColor = Color.Blue;
richTextBox1.SelectedText = "蓝" + "\n";
//将 SelectionBullet 属性置为 false, 则下面的文本将不再应用项目符号样式
richTextBox1.SelectionBullet = false;
//设置 RichTextBox 控件中的剩余内容, 以及相应的字体大小和样式
richTextBox1.SelectionFont = new Font("Arial", 16);
richTextBox1.SelectedText = "列举结束";
}

```

运行结果如下图所示。图 13.8 是在没有单击按钮前的窗体样式, 图 13.9 是单击了按钮后的窗体样式。

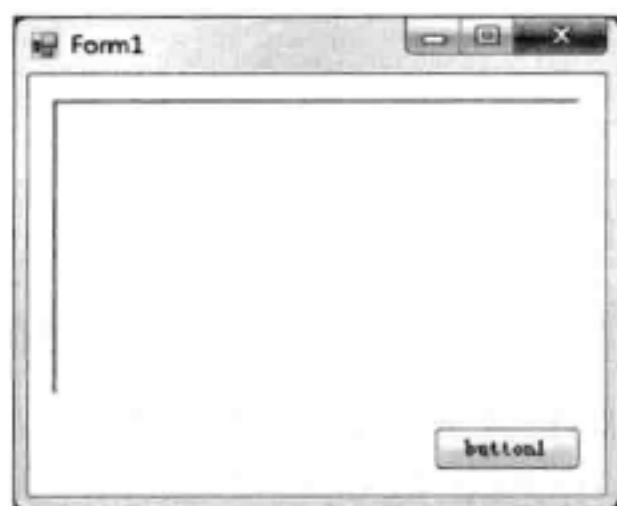


图 13.8 RichTextBox 控件

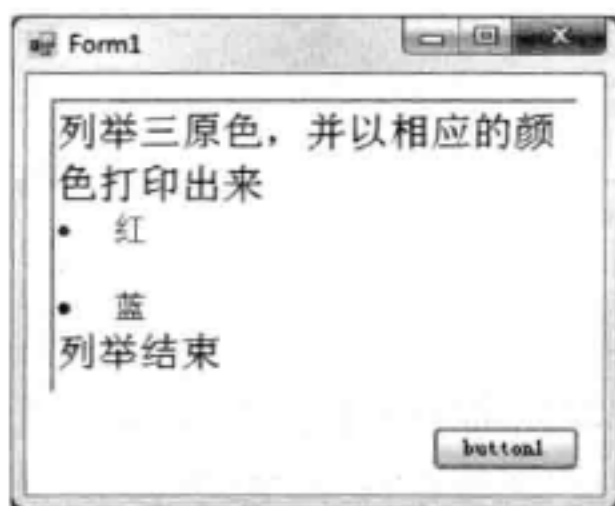


图 13.9 RichTextBox 控件的项目符号样式

13.1.12 设置 RichTextBox 控件的文本缩进

SelectionIndent 属性是一个 int 类型的对象, 它表示在 RichTextBox 控件中当前选定文本的左边缩进距离。如果 RichTextBox 控件中没有选定的文本, 则 SelectionIndent 属性所

设置的缩进距离将会对当前插入点所在的段落及后面输入的内容起作用。具体设置代码如下所示：

```
private void SetSelectionIndent()
{
    //清空 RichTextBox 控件中的内容
    richTextBox1.Clear();
    //设置 SelectionIndent 属性的值为 20 像素
    richTextBox1.SelectionIndent = 20;
    //向 RichTextBox 控件中添加内容
    richTextBox1.SelectedText = "将设置 SelectionIndent 属性的值为 20 像素的文本\n";
    richTextBox1.SelectionIndent = 0;
    richTextBox1.SelectedText = "将设置 SelectionIndent 属性的值为 0 像素的文本";
}
```

程序的运行结果如图 13.10 所示。

SelectionHangingIndent 属性是一个 int 类型的对象，它代表在 RichTextBox 控件中的当前选定文本的悬挂缩进距离。如果 RichTextBox 控件中没有选定的文本，则该属性所设置的悬挂缩进距离将会对当前插入点所在的段落及后面输入的内容起作用。具体设置代码如下所示。

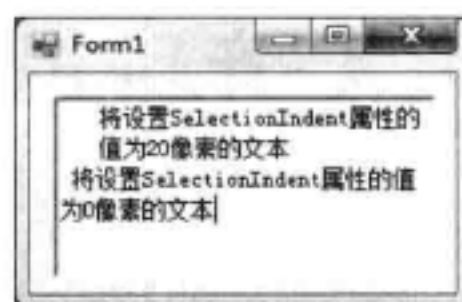


图 13.10 SelectionIndent 属性

```
private void SetSelectionHangingIndent()
{
    //清空 RichTextBox 控件中的内容
    richTextBox1.Clear();
    //设置 SelectionHangingIndent 属性的值为 20 像素
    richTextBox1.SelectionHangingIndent = 20;
    //向 RichTextBox 控件中添加内容
    richTextBox1.SelectedText = "将设置 SelectionHangingIndent 属性的值为 20 像素的文本\n";
    richTextBox1.SelectionHangingIndent = 0;
    richTextBox1.SelectedText = "将设置 SelectionHangingIndent 属性的值为 0 像素的文本";
}
```

程序的运行结果如图 13.11 所示。

SelectionRightIndent 属性是一个 int 类型的对象，它表示对 RichTextBox 控件中当前选定文本的右边的缩进距离。如果 RichTextBox 控件中没有选定的文本，则该属性所设置的缩进距离将会对当前插入点所在的段落及后面输入的内容起作用。具体设置代码如下所示。

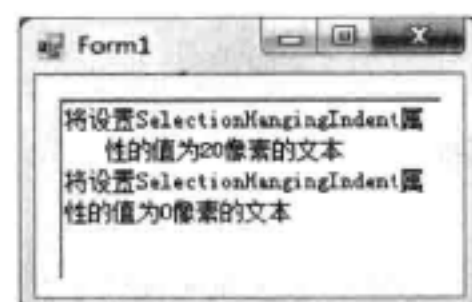


图 13.11 SelectionHangingIndent 属性


```
private void SetSelectionRightIndent()
{
    //清空 RichTextBox 控件中的内容
    richTextBox1.Clear();
    //设置 SelectionRightIndent 属性的值为 20 像素
```

```

richTextBox1.SelectionRightIndent= 20;
//向 RichTextBox 控件中添加内容
richTextBox1.SelectedText = "将设置 SelectionRightIndent 属性的值
为 20 像素的文本\n";
richTextBox1.SelectionRightIndent= 0;
richTextBox1.SelectedText = " 将设置 SelectionRightIndent 属性的值为 0 像素
的文本";
}

```

程序的运行结果如图 13.12 所示。

说明：通过对以上 3 个属性进行设置，可以完成各种形式的文本缩进。

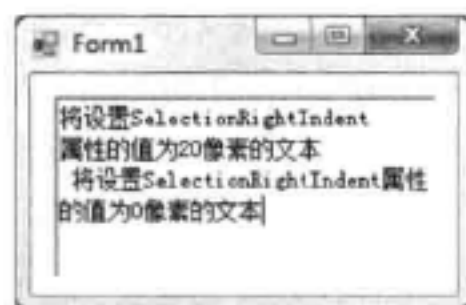


图 13.12 SelectionRightIndent 属性

13.1.13 在 RichTextBox 控件中添加超链接

DetectUrls 属性是一个 boolean 类型的变量，它指示了在 RichTextBox 控件中是否对 URL 设置特殊的超链接格式，当 DetectUrls 属性为 true 时，可以将 Web 链接显示为彩色或下划线形式的超链接格式。另外，也可以通过编写代码，在单击链接时打开浏览器窗口，该窗口中显示链接文本中指定的网站。

在下面的示例中，将通过对 RichTextBox 控件的相关属性进行设置来完成单击 URL 而打开相应网页的功能。


首先，将 Text 属性设置为包含有效的 URL（例如 <http://www.microsoft.com>），并确保将 DetectUrls 属性设置为 true。然后，创建一个 Process 对象的实例，用以启动一个进程能够打开 IE 浏览器。最后，为 LinkClicked 事件编写事件处理程序，将所需的文本发送到浏览器。具体设置代码如下所示。

```

public System.Diagnostics.Process p = new System.Diagnostics.Process();
private void richTextBox1_LinkClicked(object sender, System.Windows.Forms.
LinkClickedEventArgs e)
{
    //启动进程，打开下划线所指示的链接地址的网站
    p = System.Diagnostics.Process.Start("IExplore.exe", e.LinkText);
}

```

LinkClicked 事件将在单击 RichTextBox 控件中的 URL 时发生。

说明：System.Diagnostics.Process.Start()方法是开始一个进程的方法。

13.1.14 RichTextBox 控件的文件操作


在 RichTextBox 控件中还可以使用 LoadFile()方法从流加载数据。在下面的示例中，假设系统中已经存在一个“C:\test.rtf”。如果在调用 LoadFile()方法时仅使用文件名作为其唯一参数，则会假定该文件为 RTF 文件。若要指定其他文件类型，则以 RichTextBoxStreamType

枚举的值作为其第二个参数来调用该方法。然后，在 RichTextBox 控件中打开并显示选定的文件。代码设置如下：

```
private void button1_Click(object sender, EventArgs e)
{
    //加载文档"C:\test.rtf"
    richTextBox1.LoadFile(@"C:\test.rtf",
        RichTextBoxStreamType.RichText);
}
```

在 RichTextBox 控件中还可以使用 SaveFile()方法将数据保存到流中。在下面的示例中，假设要保存的文件路径和文件名为“C:\test.rtf”。然后调用 SaveFile()方法对文件进行保存，如果调用该方法时，仅使用文件名作为其唯一参数，则该文件将保存为 RTF 文件。代码设置如下：

```
private void button2_Click(object sender, EventArgs e)
{
    //保存文本框中的文本为"C:\test.rtf"文档
    richTextBox1.SaveFile(@"C:\test.rtf");
}
```

 **技巧：**若要指定其他文件类型，则通过以 RichTextBoxStreamType 枚举的值作为其第二个参数来调用 SaveFile()方法。在本例中，RTF 文件的枚举值为 RichTextBoxStreamType.RichNoOleObjs。

在 RichTextBox 控件中还可以使用 SaveFile 方法将数据保存到数据流中。在下面的示例中，假设要保存的文件路径和文件名为“C:\test.rtf”。然后调用 SaveFile 方法对文件进行保存，如果调用该方法时仅使用文件名作为其唯一参数，则该文件将保存为 RTF 文件。若要指定其他文件类型，则通过以 RichTextBoxStreamType 枚举的值作为其第二个参数来调用该方法。代码设置如下：

```
private void button2_Click(object sender, EventArgs e)
{
    richTextBox1.SaveFile(@"C:\test.rtf", RichTextBoxStreamType.Rich-
        NoOleObjs);
}
```

在 RichTextBox 控件中还可以使用 Find()方法搜索指定的字符串。在下面的示例中，将搜索指定的字符串，并返回指定的字符串的起始索引号。

```
private void button3_Click(object sender, EventArgs e)
{
    //搜索文本框中的指定的文本
    string text = this.textBox1.Text;
    //返回指定文本在 richTextBox1 中的索引
    int indexToText = richTextBox1.Find(text );
}
```

13.2 不能编辑的文本显示控件

在 Visual Studio 中, 文本显示控件主要包括显示用户无法直接编辑文本的 Label 控件; 使文本显示为 Web 样式链接的 LinkLabel 控件; 向用户显示操作当前进度的 ProgressBar 控件和向用户显示有关应用程序的当前状态信息的 StatusStrip 控件。

13.2.1 用 Label 控件显示文本

Label 控件在 .NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中, Label 控件主要用于提供描述性语言, 通常与其他控件一起使用, 用于说明其他控件的用途。

Label 控件在 Visual Studio 的工具箱中的图示如图 13.13 所示, 将 Label 控件拖曳到窗体选定后, 如图 13.14 所示。然后, 可在 Visual Studio 的属性窗体中对 Label 的属性进行设置, 使其达到应用程序的具体需求。



图 13.13 Label 控件

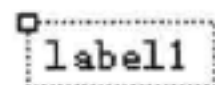


图 13.14 Label 控件

AutoSize 属性是一个 boolean 类型的变量, 使用此属性可以指示出当前 Label 控件是否根据文本里的内容自动调整 Label 控件的大小。如果 AutoSize 的属性值为 true, 则 Label 控件可以自动调整宽度来显示 Label 控件的全部文本内容。如果 AutoSize 的属性值为 false, 则 Label 控件不能自动调整宽度, 而是固定为 Label 控件的 Size 属性所指定的宽度。可以在属性窗体里对 Label 控件的 AutoSize 属性进行设置, 也可以在代码中进行设置, 具体的设置代码如下:

```
private void setAutoSize()  
{  
    //设置文本标签为自动化大小  
    this.label1.AutoSize= true;  
}
```

当 AutoSize 属性为 false 时, 如果 Label 控件中的文本长度超过了 Label 控件的宽度, 则可以通过设置 AutoEllipsis 属性来优化用户界面的视觉效果。AutoEllipsis 属性是一个 boolean 类型的变量, 使用此属性可以指示出当 Label 控件里的文本超出了 Label 控件的长度时, Label 控件是否会自动在部分可显示出的文本后面添加省略号。如果 AutoEllipsis 的属性值为 true, 则可以通过省略号指示存在尚未显示的文本。可以在属性窗体里对 Label 控件的 AutoEllipsis 属性进行设置, 也可以在代码中进行设置, 具体的设置代码如下:

```
private void setAutoEllipsis()  
{  
    //设置文本标签为可自动优化, 添加省略号  
    //设置 AutoEllipsis 属性为 true 时, AutoSize 属性为 false
```



```
this.label1.AutoEllipsis= true;
}
```

FlatStyle 属性将返回一个 FlatStyle 类型的枚举值,用以指示 Label 控件的外观。FlatStyle 的具体成员列表如表 13.2 所示。

表 13.2 FlatStyle 成员列表


成 员	说 明
Flat	平面外观
Popup	正常情况下为平面外观,当有鼠标经过时,改为三维外观
Standard	三维外观
System	由操作系统决定的外观

可以在属性窗体里对 Label 控件的 FlatStyle 属性进行设置,也可以在代码中进行设置,具体的设置代码如下:

```
private void setFlatStyle()
{
    //设置文本标签的外观为鼠标经过时为三维外观
    this.label1.FlatStyle= Popup;
}
```

13.2.2 用 LinkLabel 控件显示超链接文本

LinkLabel 控件在 .NET 中的命名空间为 System.Windows.Forms。在 Windows 应用程序编程中,LinkLabel 控件是一种可提供超链接的标签控件。

说明: LinkLabel 控件除了可显示超链接外,其他功能与 Label 控件相似。

LinkLabel 控件在 Visual Studio 工具箱中的图示如图 13.15 所示,将 Label 控件拖曳到窗体选定后,如图 13.16 所示。然后,可在 Visual Studio 的属性窗体中对 LinkLabel 的属性进行设置,使其达到应用程序的具体需求。

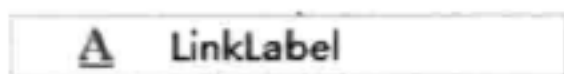


图 13.15 LinkLabel 控件

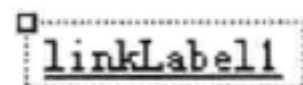


图 13.16 LinkLabel 控件

LinkColor 属性返回一个 Color 类型的属性值,用以显示 LinkLabel 控件中所有链接的原始颜色。ActiveLinkColor 属性返回一个 Color 类型的属性值,用以显示当链接是活动时的颜色。DisabledLinkColor 属性也返回一个 Color 类型的属性值,用以显示当链接不能使用时的颜色。LinkLabel 控件中的链接,对以上 3 个属性的设置可以在控件的属性窗体中进行设置,也可以通过以下代码进行设置。

```
private void setLinkLabelColor()
{
    //设置文本标签的超链接部分的颜色
    this.linkLabel1.ActiveLinkColor = System.Drawing.Color.Maroon;
}
```

```

        this.linkLabel1.DisabledLinkColor = System.Drawing.Color.Orange;
        this.linkLabel1.LinkColor = System.Drawing.Color.Blue;
    }

```

LinkBehavior 属性的属性值是 LinkBehavior 枚举值之一，它指定了 LinkLabel 控件中的链接行为。LinkBehavior 的具体成员列表如表 13.3 所示。

表 13.3 LinkBehavior 成员列表

成 员	说 明
AlwaysUnderline	链接显示为带下划线的文本
HoverUnderline	当鼠标经过此链接时，链接显示为带下划线的文本
NeverUnderline	链接不显示为带下划线的文本
SystemDefault	链接的外观表现取决于系统的设置

可以在属性窗体里对 LinkLabel 控件的 LinkBehavior 属性进行设置，也可以在代码中进行设置，具体设置代码如下：

```

private void setLinkBehavior()
{
    //设置文本标签的超链接部分的超链接行为
    this.linkLabel1.LinkBehavior = System.Windows.Forms.LinkBehavior.
    AlwaysUnderline;
}

```

LinkVisited 属性是一个 boolean 型的变量，用以显示此链接是否被访问过，如果被访问过则 LinkVisited 的属性值为 true。另外也可以通过将 LinkVisited 属性值设置为 true，使 LinkLabel 控件的链接部分显示为被访问过的外观。具体代码如下：

```

private void setLinkVisited()
{
    //设置超链接文本是否已被访问过
    this.linkLabel1.LinkVisited= true;
}

```

LinkArea 属性用来标志在 LinkLabel 控件中的文本的链接区域，如果在 LinkLabel 控件中只有一个超链接，则可以使用 LinkArea 属性来确定。LinkArea 属性值是一个 LinkArea 结构的变量。可以通过对 LinkArea 结构的相关属性设置来指定 LinkLabel 控件中的文本超链接区域。LinkArea 结构的两个主要属性分别是 Length 属性和 Start 属性。其中 Start 属性指定了 LinkLabel 控件中的文本超链接区域的起始位置，而 Length 属性则指定了超链接区域包含的字符数。将 Start 属性和 Length 属性结合使用，就可以准确无误的定义出 LinkLabel 控件中的文本超链接区域，具体定义方法如下所示。

```

private void setLinkVisited()
{
    //设置文本标签中含有超链接的部分
    this.linkLabel1.LinkArea = new System.Windows.Forms.LinkArea(0, 8);
}

```

 注意：LinkArea 属性值是一个 LinkArea 结构的变量。

13.2.3 保存超链接的 LinkCollection 集合

Links 属性是一个 LinkCollection 类的变量。它表示在 LinkLabel 控件中的文本链接的集合。LinkCollection 类是 Link 类对象的集合。

Link 类表示的是 LinkLabel 控件中的一个超链接，它的主要属性有 Length 属性、Start 属性、LinkData 属性和 Name 属性。其中最为主要的还是 Start 属性和 Length 属性，其用法和用途与 LinkArea 结构中的 Start 属性和 Length 属性一致。

在 LinkCollection 类中，可以使用 Add 方法向 LinkCollection 集合中添加新的 Link 对象。使用 Clear 方法清除 LinkCollection 集合中的所有超链接对象。使用 Remove 方法从 LinkCollection 集合中移除超链接对象。使用 RemoveAt 方法从 LinkCollection 集合中的指定位置移除超链接对象。

13.2.4 LinkLabel 控件编程示例

下面将通过一个例子来具体讲解 LinkLabel 控件的属性和事件在具体编程中的使用方法。在此例中，将在窗体中设置一个 LinkLabel 控件，在其上添加描述性文字，并在文件的相应位置设置超链接。

1. 创建项目

创建一个 Windows 应用程序，命名为“13.2.4”。程序自动生成了一个 Form。修改 Form 的属性，如表 13.4 所示。

表 13.4 Form 属性设置

属 性	值	属 性	值
Size	300, 100	Text	LinkLabel Example

2. 设置属性和事件

向窗体添加 LinkLabel 控件，并设置 LinkLabel 控件的属性，如表 13.5 所示。

表 13.5 LinkLabel 属性设置

属 性	值
AutoSize	False
DisabledLinkColor	Red
LinkArea	0, 2
LinkBehavior	HoverUnderLine
LinkColor	Purple
Location	10, 10
Size	180, 50
Text	欢迎进入 C#编程的世界更多帮助请查阅 MSDN 了解新产品信息请访问微软主页
VisitedLinkColor	Blue

双击窗体控件，.NET 自动为窗体生成 Form1_Load 事件，如下所示。

```
private void Form1_Load(object sender, EventArgs e)
{
}
```

双击 linkLabel1，.NET 自动为窗体生成 linkLabel1_LinkClicked 事件，如下所示。

```
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClicked
EventArgs e)
{
}
```

3. 添加事件处理程序

向 Form1_Load 事件添加事件处理程序，设置 linkLabel1 里的链接对象。第一个链接对象在属性设置中已经定义，在此处只设置它的链接数据，然后向 linkLabel1 添加其他链接对象，具体设置代码如下：

```
private void Form1_Load(object sender, EventArgs e)
{
    linkLabel1.Links[0].LinkData = "欢迎进入 C#编程的世界";
    //添加链接对象，在“MSDN”处添加链接，并设置超链接数据
    linkLabel1.Links.Add(26, 4, "http://msdn2.microsoft.com/zh-cn/
default.aspx");
    //添加链接对象，在“微软”处添加链接，并设置超链接数据
    linkLabel1.Links.Add(49, 2, "www.microsoft.com");
    //将第二个链接（MSDN）设为不可访问，显示为红色，且在用户单击时没有反应
    linkLabel1.Links[1].Enabled = false;
}
```

向 linkLabel1_LinkClicked 事件添加事件处理程序。当链接被单击时，首先确定在 LinkLabel 控件中被单击的链接，并将此链接的 Visited 属性设置为 true。因为 linkLabel1 的 VisitedLinkColor 属性为 Blue，所以单击后此链接的颜色变为蓝色。获得此链接的链接数据，简单判断链接数据是否为 URL 地址，如果是，则打开相应的链接，如果不是，则弹出一个消息框，显示这个链接的链接数据。代码如下：

```
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClicked-
EventArgs e)
{
    //将被单击的链接的 Visited 属性设为 true，则此链接的颜色变为蓝色
    linkLabel1.Links[linkLabel1.Links.IndexOf(e.Link)].Visited = true;
    //基于链接对象的 LinkData 属性的值，或取相应的链接
    string target = e.Link.LinkData as string;
    //如果这个链接看起来像一个 URL 地址，访问这个地址，否则，在消息框中显示这个链接
    if (null != target && target.StartsWith("www"))
    {
        //启动进程，打开指定的网页
        System.Diagnostics.Process.Start(target);
    }
    else
    {
    }
```



```

{
    //弹出消息对话框
    MessageBox.Show(target);
}
}

```

4. 编译并运行程序

对项目“13.2.4”进行编译，运行结果如图 13.17 和图 13.18 所示。

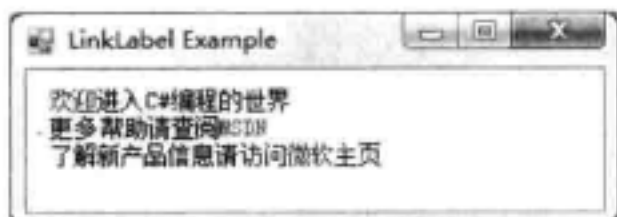


图 13.17 程序运行结果



图 13.18 程序运行结果

13.3 包容其他控件的容器控件

容器控件就是一种能够包括其他控件的控件，它可以像一般的容器一样存放其他的控件，在.NET 中常用的容器控件主要有 Panel 控件、GroupBox 控件、TabControl 控件和 TableLayoutPanel 控件这几种。

13.3.1 用 Panel 控件细分窗体

Panel 控件在.NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中，Panel 控件用于为其他控件提供可识别的分组。

Panel 控件在 Visual Studio 的工具箱中的图示如图 13.19 所示，将 Panel 控件拖曳到窗体选定后，如图 13.20 所示。然后，可在 Visual Studio 的属性窗体中对 Panel 控件的属性进行设置，使其达到应用程序的具体需求。



图 13.19 Panel 控件



图 13.20 Panel 控件

说明：Panel 控件作为容器控件的一种，可以在其中包含其他的控件。通过 Panel 控件，可以将 Windows 窗体进一步细分。

在 Windows 窗体应用程序中，用户可以通过按键盘上的 Tab 键来激活控件，控件获得焦点的顺序由 TabIndex 属性确定。另外，也可由 TabStop 属性来指示应用程序，此控件是否可通过按 Tab 键来获得焦点。对于 TabIndex 属性和 TabStop 属性均可以通过属性窗体进

行设置，也可以通过代码进行设置，具体设置代码如下：

```
private void setTabProperty()
{
    //指示 Panel 控件是否可通过 Tab 键被激活
    this.Panel1.TabIndex=1;
    this.Panel1.TabStop=false;
}
```

对于 Panel 控件以及所有的容器控件而言，都有两个属性在初始化时要认真设置，这两个属性分别是 Dock 属性和 Anchor 属性。

Anchor 属性定义了控件的父控件大小发生变化时，控件的大小将发生何种变化。Anchor 的属性值是 AnchorStyles 枚举值的组合，AnchorStyles 的具体成员列表如表 13.6 所示。

表 13.6 AnchorStyles 成员列表

成 员	说 明
None	当调整父控件的大小时控件的大小保持不变
Top	当调整父控件的大小时控件的上边缘与父控件的边缘的相对位置保持不变
Bottom	当调整父控件的大小时控件的下边缘与父控件的边缘的相对位置保持不变
Left	当调整父控件的大小时控件的左边缘与父控件的边缘的相对位置保持不变
Right	当调整父控件的大小时控件的右边缘与父控件的边缘的相对位置保持不变

可以在 Windows 属性窗体中对 Panel 控件的 Anchor 属性进行设置，也可以在代码中对其进行设置，使用代码设置的方法如下：

```
private void setPanelAnchor()
{
    //指示 Panel 控件的锚定方式
    this.Panel1.Anchor=(AnchorStyles.Bottom | AnchorStyles.Right);
}
```

Dock 属性同样也定义了控件的父控件大小发生变化时，控件的大小将发生何种变化。不同的是，Dock 的属性值是 DockStyle 枚举值的组合。DockStyle 的具体成员列表如表 13.7 所示。

表 13.7 DockStyle 成员列表

成 员	说 明
None	控件未停靠在父控件上
Top	控件的上边缘停靠在父控件的上端
Bottom	控件的下边缘停靠在父控件的下端
Left	控件的左边缘停靠在父控件的左端
Right	控件的右边缘停靠在父控件的右端
Fill	控件的各边缘停靠在父控件的各端

可以在 Windows 属性窗体中对 Panel 控件的 Dock 属性进行设置，也可以在代码中对其进行设置，使用代码设置的方法如下：


```
private void setPanelDock()
{
    //指示 Panel 控件的停靠方式
    this.Panel1.Dock=DockStyle.Right;
}
```


当 Panel 控件的 Anchor 属性和 Dock 属性都被设置的时候，将按照后设置的属性所设置的行为执行。

在 Windows 应用程序开发中，有时候需要根据前面所进行的操作在代码中动态生成 Windows 窗体，而不是一开始的窗体设计器中对控件的位置、外观、行为，以及属性进行设置。下面的代码示例创建了一个 Panel 控件，并向 Panel 添加一个 Label 控件和一个 TextBox 控件。为了使读者更好地观察程序的运行效果，这里的 Panel 控件将显示为三维边框。

```
public void CreateMyPanel()
{
    //创建窗体中加载的控件
    Panel panel1 = new Panel();
    TextBox textBox1 = new TextBox();
    Label label1 = new Label();
    //初始化 Panel 控件
    panel1.Location = new Point(50, 50);
    panel1.Size = new Size(250, 150);
    //Panel 控件显示为三维边框
    panel1.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
    //相对于 Panel 控件的位置
    label1.Location = new Point(15,15);
    label1.Text = "label1";
    label1.Size = new Size(100, 15);
    //相对于 Panel 控件的位置
    textBox1.Location = new Point(15,30);
    textBox1.Text = "textBox1";
    textBox1.Size = new Size(150, 20);
    //向窗体中添加控件
    this.Controls.Add(panel1);
    panel1.Controls.Add(label1);
    panel1.Controls.Add(textBox1);
}
```

AutoScroll 属性是一个 boolean 类型的值，指示当 Panel 控件中包含的子控件超过了 Panel 控件的大小时，是否自适应显示滚动条。当 AutoScroll 属性为 true 时，将自适应显示滚动条。默认情况下，其值为 false，不会自动产生滚动条，可以通过代码对 AutoScroll 属性进行设置。具体设置代码如下：


```
private void setAutoScroll()
{
    //设置面板是否自动显示滚动条
    this.Panel1.AutoScroll=true;
}
```

 **注意：** Panel 控件并没有 Text 属性，也就是说 Panel 控件并不显示标题项。如果需要显示标题项的容器控件，则可以使用将在 13.3.2 节中介绍的 GroupBox 控件。

13.3.2 有标题的容器控件 GroupBox

Windows 窗体的 GroupBox 控件与在 13.3.1 节中介绍的 Panel 控件功能相同，都是用于为其他控件提供可识别的分组控件，都属于容器控件。因此在设计阶段，无论是移动 GroupBox 控件还是 Panel 控件，其所包含的控件都将随之一起移动，各控件间的相对距离保持不变。

当然，Panel 控件和 GroupBox 控件之间也存在着差别，例如 GroupBox 控件可以显示标题，而 Panel 控件的 Text 属性却没有实际意义。Panel 控件在其所包含的子控件超出其可视范围时，可以自动生成自适应的滚动条，而 GroupBox 控件不可以。

 **注意：** 只有 GroupBox 控件中的子控件才可以被激活，GroupBox 控件是不会接收焦点的。

GroupBox 控件在 .NET 中命名空间为 System.Windows.Forms。GroupBox 控件在 Visual Studio 的工具箱中的图示如图 13.21 所示，将 GroupBox 控件拖曳到窗体并选定后，如图 13.22 所示。然后，可在 Visual Studio 的属性窗体中对 GroupBox 控件的属性进行设置，使其达到应用程序的具体需求。

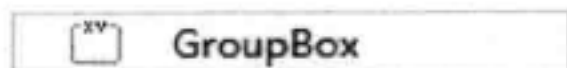


图 13.21 GroupBox 控件



图 13.22 GroupBox 控件

可以通过 GroupBox 控件的 Text 属性来设置 GroupBox 控件的标题，通过此标题可以显示出此分组的主要业务功能。也可以在属性窗体中对此属性进行设置，还可以通过代码对此属性进行设置，具体设置代码如下：

```
private void setGroupBoxText(string text)
{
    this.GroupBox.Text=text;
}
```

下面的代码示例创建一个 GroupBox 控件，并向该控件添加一个 Label 控件和一个 TextBox 控件，并将该 GroupBox 组框添加到 Form 中。

```
public void InitializeMyGroupBox()
{
    //创建窗体中加载的控件
    GroupBox groupBox1 = new GroupBox();
    TextBox textBox1 = new TextBox();
    Label label1 = new Label();
```



```

//初始化 GroupBox 控件
groupBox1.FlatStyle = FlatStyle.Flat;
groupBox1.Location = new Point(50, 50);
groupBox1.Size = new Size(250, 150);
//相对于 GroupBox 控件的位置
label1.Location = new Point(15,15);
label1.Text = "label1";
label1.Size = new Size(100, 15);
//相对于 GroupBox 控件的位置
textBox1.Location = new Point(15,30);
textBox1.Text = "textBox1";
textBox1.Size = new Size(150, 20);
//向窗体中添加控件
groupBox1.Controls.Add(label1);
groupBox1.Controls.Add(textBox1);
this.Controls.Add(groupBox1);
}

```

13.3.3 多页容器控件 TabControl

TabControl 控件在 .NET 中的命名空间为 System.Windows.Forms。在 Windows 应用程序编程中, TabControl 控件主要用于生成多页对话框, 每一页都可以理解成一个容器控件。

TabControl 控件在 Visual Studio 的工具箱中的图示如图 13.23 所示, 将 TabControl 控件拖曳到窗体选定后, 如图 13.24 所示。然后, 可在 Visual Studio 的属性窗体中对 TabControl 的属性进行设置, 使其达到应用程序的具体需求。

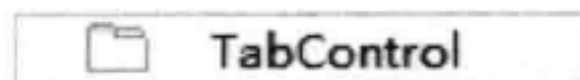


图 13.23 TabControl 控件

如图 13.24 所示, 在 TabControl 控件中可以显示多个 TabPage (在本例中是两个), 为了便于理解, 可以将 TabControl 控件想象成 TabPage 的父控件或容器, 而 TabPage 才是其他用于完成业务功能的子控件的容器。在图 13.24 中, TabControl 控件为被激活状态, 这时, 可以通过属性窗体对 TabControl 控件的属性进行具体的定制, 以满足系统的具体要求。而在图 13.25 中, 则是 TabControl 控件的子控件 tabPage1 为被激活时的状态。这时, 可以通过属性窗体对 tabPage1 控件的属性进行具体的定制, 以满足系统的具体要求。

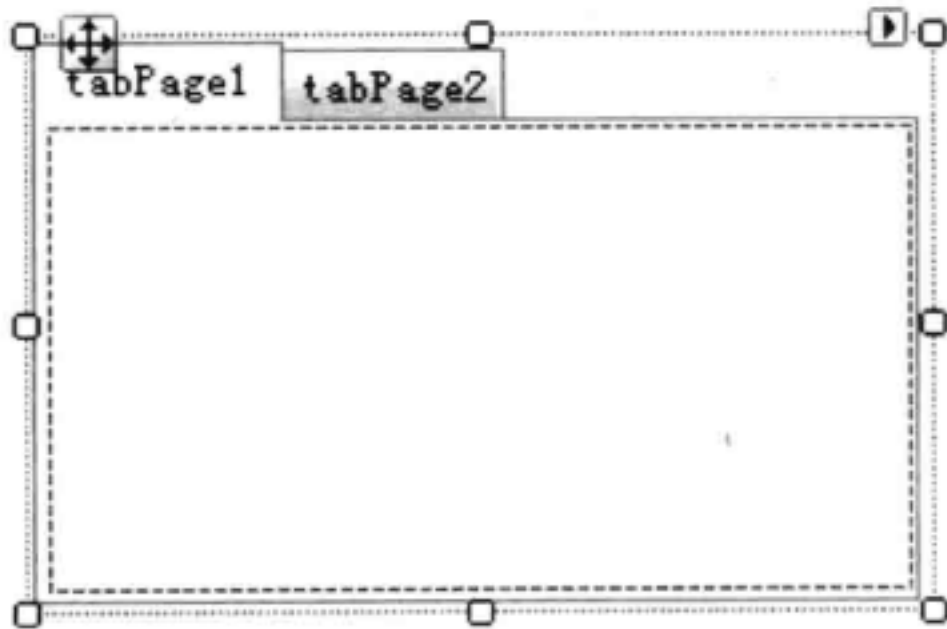


图 13.24 TabControl 控件

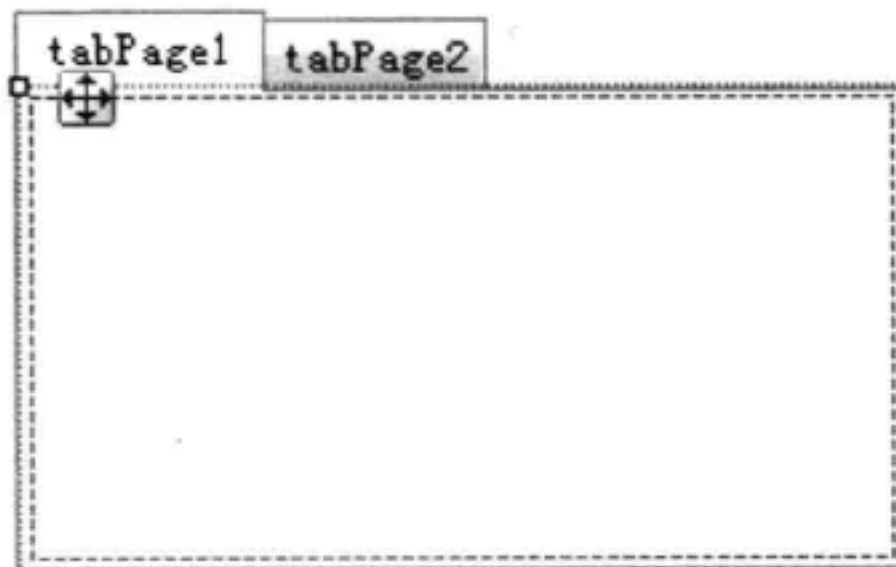


图 13.25 TabControl 控件

⚠注意: TabControl 控件只是 TabPage 的容器, 而 TabPage 才是承载各子控件的容器。

13.3.4 TabControl 控件的几种常见外观

Alignment 属性可以用来设置在 TabControl 控件中 TabPage 的排列方式。默认情况下, TabPage 以并排的方式排列在 TabControl 控件的上方, 如图 13.24 所示, 但是通过对 Alignment 属性的设定, 可以改变 TabPage 的标签头在 TabControl 控件中出现的位置, 如图 13.26 和图 13.27 所示。



图 13.26 TabControl 控件

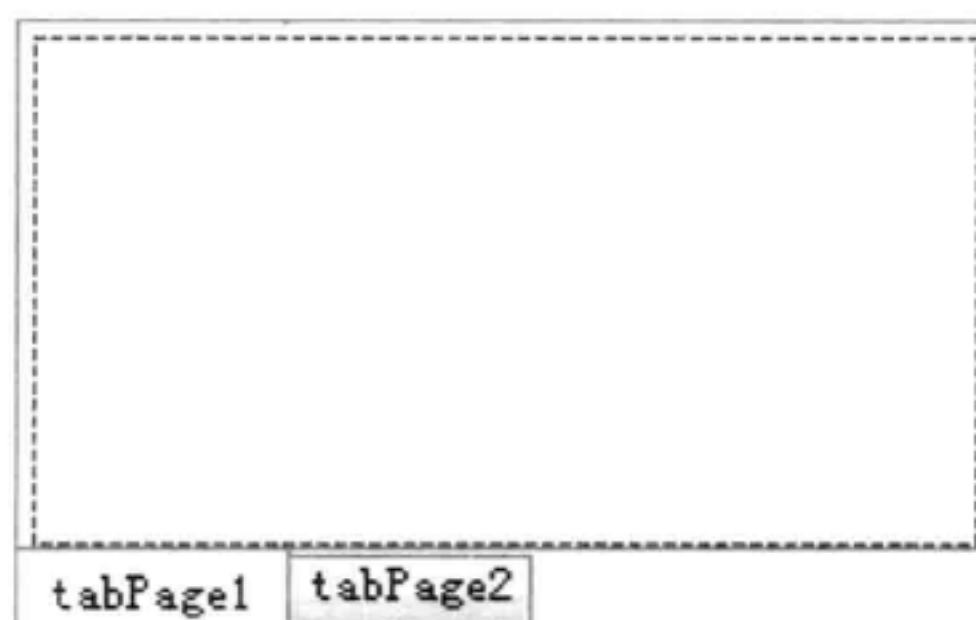


图 13.27 TabControl 控件

Alignment 属性值是一个 TabAlignment 枚举类型。TabAlignment 的具体成员列表如表 13.8 所示。

表 13.8 TabAlignment 成员列表

成 员	说 明
Top	TabPage 的标签头在 TabControl 控件的顶部
Bottom	TabPage 的标签头在 TabControl 控件的底部
Left	TabPage 的标签头在 TabControl 控件的左侧
Right	TabPage 的标签头在 TabControl 控件的右侧

可以在设计时通过属性窗体对 TabControl 控件的 Alignment 属性进行设置, 也可以在代码中对其进行设置。使用代码设置 Alignment 属性的方法如下:

```
private void setTabControlAlignment()
{
    //设置 TabControl 控件中 TabPage 标签的显示位置
    this.tabControl1.Alignment = System.Windows.Forms.TabAlignment.Bottom;
}
```

Appearance 属性可以用来设置在 TabControl 控件中 TabPage 标签头的显示样式。默认情况下, TabPage 标签头的显示样式如图 13.24 所示, 但是通过对 Appearance 属性的设定, 可以改变 TabPage 的标签头的显示, 如图 13.28 和图 13.29 所示。

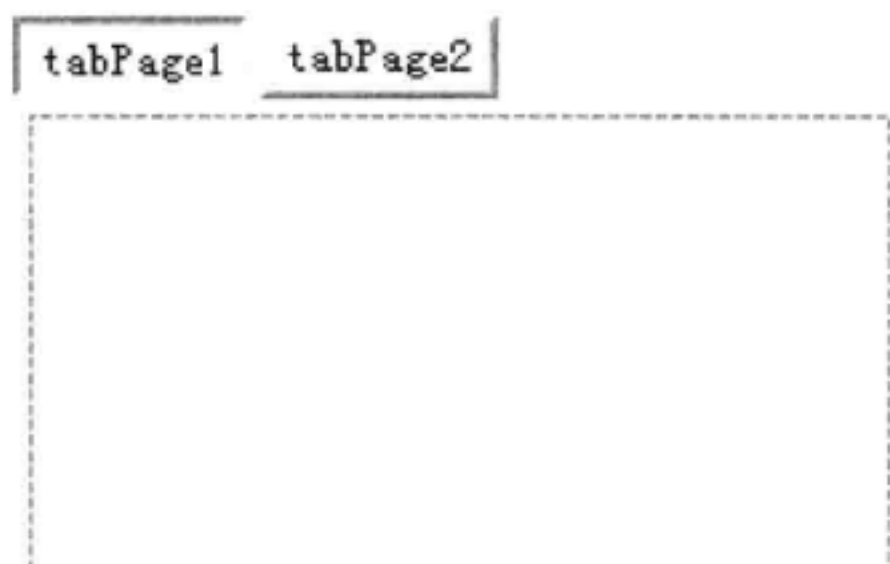


图 13.28 TabControl 控件

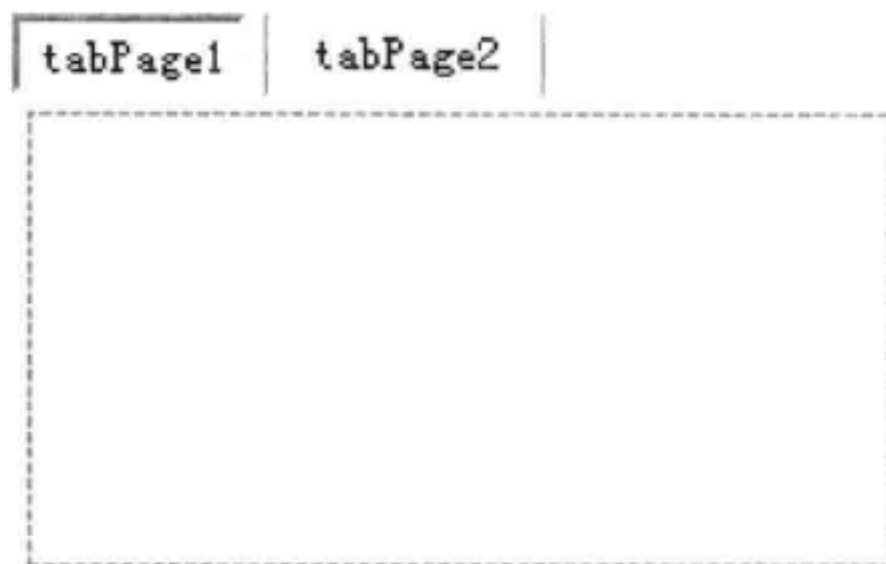


图 13.29 TabControl 控件

Appearance 属性值是一个 TabAppearance 枚举类型, TabAppearance 的具体成员列表如表 13.9 所示。

表 13.9 TabAppearance 成员列表

成 员	说 明
Normal	TabPage 标签头的外观类似于笔记本上的标签页, 如图 13.24 所示
Buttons	TabPage 标签头的外观类似于 Button 控件, 如图 13.28 所示
FlatButtons	TabPage 标签头的外观类似于平面按钮, 如图 13.29 所示

 注意: Appearance 属性用来设置 TabControl 控件中的 TabPage 标签头的显示样式。

可以在设计时通过属性窗体对 TabControl 控件的 Appearance 属性进行设置, 也可以在代码中对其进行设置。使用代码设置的方法如下:

```
private void setTabControlAlignment()
{
    //设置 TabControl 控件中 TabPage 标签的外观
    this.tabControl1.Appearance = System.Windows.Forms.TabAppearance.Buttons;
}
```

Multiline 属性可以用来设置在 TabControl 控件中是否能以多行方式排列 TabPage 的标签头。在应用程序设计中, 会遇见一个 TabControl 控件中包含多个 TabPage 的情况, 如图 13.30 所示, 当 TabPage 的标签头的宽度之和大于 TabControl 控件的宽度时, TabPage 的标签头将无法完全显示出来。在这种情况下, 标签头的右侧通常会出现一对左右箭头, 用户可以通过单击箭头而移动标签项, 选择想要进行操作的 TabPage。

但是若将 Multiline 属性值设置为 true, 则当 TabPage 的标签头的宽度之和大于 TabControl 控件的宽度时, TabPage 的标签头将以超过一行的方式显示出来。如图 13.31 所示, 这样设置后, 用户就可以看到所有的标签页了。

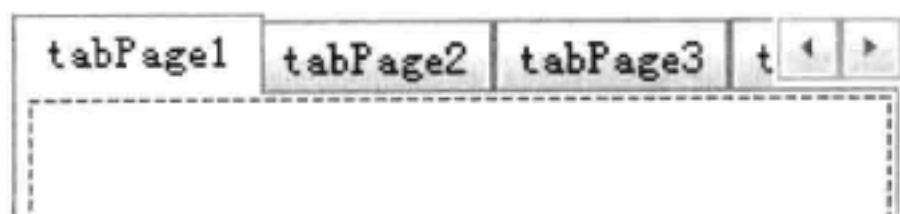


图 13.30 TabControl 控件

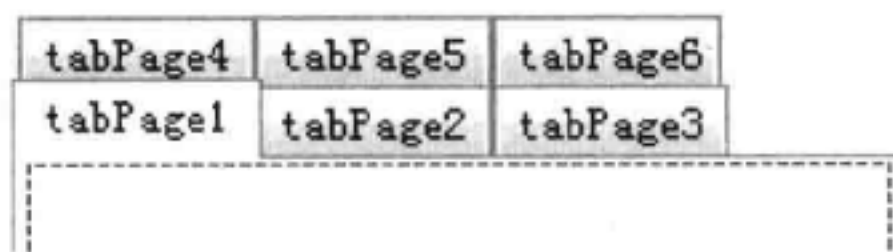


图 13.31 TabControl 控件

可以在设计时通过属性窗体对 TabControl 控件的 Multiline 属性进行设置,也可以在代码中对其进行设置。使用代码设置的方法如下:

```
private void setTabControlMultiline()  
{  
    //设置 TabControl 控件中 TabPage 标签是否可多行显示  
    this.tabControl1.Multiline= true;  
}
```


当 Multiline 属性值设置为 true 时,可以通过 RowCount 属性获得当前 TabControl 控件中 TabPage 的标签头的行数。另外,还可以通过 TabCount 属性获得当前 TabControl 控件中 TabPage 的个数。可以使用下面的代码获得程序运行需要的信息。

```
private void getTabPageNum()  
{  
    int rowCount = 0;  
    int tabCount = 0;  
    rowCount =this.tabControl1.RowCount;  
    tabCount =this.tabControl1.TabCount;  
}
```

SizeMode 属性指示出了确定 TabControl 控件中的 TabPage 标签头的宽度的方法。SizeMode 属性值是一个 TabSizeMode 枚举类型, TabSizeMode 的具体成员列表如表 13.10 所示。

表 13.10 TabSizeMode成员列表

成 员	说 明
Normal	TabPage 标签头的宽度自适应于 TabPage 标签头的内容
Fixed	TabControl 控件中的各 TabPage 的宽度一致
FillToRight	当 TabControl 控件的 Multiline 属性值置为 true 时, 每行 TabPage 标签头宽的宽度等于 TabControl 控件的宽度

 **技巧:** 当 SizeMode 属性值设置为 TabSizeMode.Fixed 时, 可以通过 ItemSize 属性设置 TabPage 标签头的宽度。

可以在设计时通过属性窗体对 TabControl 控件的 SizeMode 属性和 ItemSize 属性进行设置,也可以在代码中对其进行设置。使用代码设置的方法如下:

```
private void setTabControlItemSize()  
{  
    this.tabControl1.SizeMode=TabSizeMode.Fixed;  
    this.tabControl1.ItemSize =80;  
}
```

13.3.5 TabControl 控件中标签页的选择

在应用程序运行时,用户可以通过单击 TabControl 控件中的 TabPage 标签头来切换不同的标签页。同样,也可以通过在程序中对 TabControl 控件的属性进行设置来选择当前的

标签页，具体的相关属性分别是 `SelectedIndex` 属性和 `SelectedTab` 属性。


对 `SelectedIndex` 属性进行设置，就是通过设置希望激活的标签页的索引来选择标签页。而对 `SelectedTab` 属性进行设置，则是通过设置希望激活的标签页本身来选择标签页。在应用程序中实现这一功能的代码如下所示。

❑ 对 `SelectedIndex` 属性进行设置：

```
private void setSelectedIndex()
{
    //激活第二个选项卡
    this.tabControl1.SelectedIndex=1;
}
```

❑ 对 `SelectedTab` 属性进行设置：

```
private void setSelectedTab()
{
    //激活第二个选项卡
    this.tabControl1.SelectedTab=this.tabPage2;
}
```

 **技巧：**也可以通过使用 `TabControl` 控件的 `DeselectTab` 方法和 `SelectTab` 方法选择当前的标签页。使用 `DeselectTab` 方法，可以激活由 `DeselectTab` 方法的参数所指定的标签页后面的标签页。

使用 `TabControl` 控件的 `DeselectTab` 方法可使指定的选项卡后面的选项卡成为当前选项卡，使用 `SelectTab` 方法可使指定的选项卡成为当前选项卡。具体使用方法如下所示。

❑ 使具有指定索引的选项卡后面的选项卡成为当前选项卡，代码如下：

```
public void DeselectTab(int tabPageIndex)
```

❑ 使具有特定名称的选项卡后面的选项卡成为当前选项卡，代码如下：

```
public void DeselectTab(string tabPageName)
```

❑ 使指定的 `TabPage` 后面的选项卡成为当前选项卡，代码如下：

```
public void DeselectTab(TabPage tabPage)
```

使用 `SelectTab` 方法，可以激活由 `SelectTab` 方法的参数所指定的标签页。具体使用方法如下所示。

❑ 根据索引号确定标签页，代码如下：

```
public void SelectTab(int tabPageIndex)
```

❑ 根据 `TabPage` 名称确定标签页，代码如下：

```
public void SelectTab(string tabPageName)
```

❑ 参数为指定的标签页，代码如下：

```
public void SelectTab(TabPage tabPage)
```


在应用程序运行过程中如果切换的是当前处于激活状态的标签页（包括由用户操作引起和程序引起），则都将按顺序引发以下一系列的事件。

- ☐ Deselecting 事件：在取消对某个标签页的选中之前发生。
- ☐ Deselected 事件：在取消对某个标签页的选中之后发生。
- ☐ Selecting 事件：在选中某个标签页之前发生。
- ☐ Selected 事件：在选中某个标签页之后发生。

在对当前选中的标签页进行切换时，还有一个事件将被触发，就是 `SelectedIndexChanged` 事件。此事件在 `TabControl` 控件的 `SelectedIndex` 属性发生变化时被触发。可以在此事件中添加事件处理函数，使用户在切换标签页时执行某种操作。

在下面的代码中，当用户对标签页进行切换时，程序将检查用户是否具有访问被激活标签页的权限，如果有，则可以访问；如果没有，将会弹出一个对话框，提示“不能加载标签页，您没有相应的权限”。

```
//当选项卡的标签页的选择发生变化时发生
private void tabControl1_SelectedIndexChanged(object sender, System.
EventArgs e)
{
    //单选框被选中，则用户具有查阅标签页二的权限
    if ((checkBox1.Checked = true) && (tabControl1.SelectedTab == tabPage2))
    {
        tabControl1.SelectedTab = tabPage2;
    }
    //单选框未被选中，则用户不具有查阅标签页二的权限
    else if ((checkBox1.Checked = false) && (tabControl1.SelectedTab ==
tabPage2))
    {
        MessageBox.Show("不能加载标签页，您没有相应的权限");
        tabControl1.SelectedTab = tabPage3;
    }
}
```

 说明：在上面的例子中，使用 `checkbox` 控件作为权限的设定，当 `checkbox` 被选中时，用户具有访问权限，否则将不具有访问权限。

13.3.6 添加 `TabControl` 控件中的标签页

`TabControl` 控件的 `TabPage` 属性里包含了所有 `TabControl` 控件中的 `TabPage`。也可以使用 `TabControl` 控件的 `GetItems` 方法获取 `TabControl` 控件中的所有标签页，它将返回一个 `TabPage` 类的对象实例组成的数组。而 `TabControl` 控件的 `TabPage` 属性返回的则是一个 `TabControl` 控件的 `TabPageCollections` 类型的对象。

在默认情况下，每个 `TabControl` 控件有两个 `TabPage` 标签页。可以通过单击 `TabControl` 控件右上角的小箭头来启动 `TabControl` 任务对话框，通过单击“添加选项卡”和“移除选

项卡”命令来添加和移除 TabControl 控件的标签页,如图 13.32 所示。

也可以通过编程方式对 TabControl 控件中的选项卡进行添加和移除,具体代码如下所示。

❑ 添加选项卡: 使用 TabPages 属性的 Add()方法。

```
private void addTabPage()
{
    string title = "TabPage " + (tabControl1.TabCount + 1).ToString();
    TabPage myTabPage = new TabPage(title);
    tabControl1.TabPages.Add(myTabPage);
}
```

❑ 移除选项卡: 若要移除选定的选项卡, 可使用 TabPages 属性的 Remove()方法。

```
private void removeTabPage()
{
    tabControl1.TabPages.Remove(tabControl1.SelectedTab);
}
```

❑ 移除选项卡: 若要移除所有选项卡, 可使用 TabPages 属性的 Clear()方法。

```
private void clearTabPage()
{
    tabControl1.TabPages.Clear();
}
```

❑ 移除选项卡: 若要移除所有选项卡, 可使用 TabControl 属性的 RemoveAll()方法。

```
private void clearTabPage()
{
    tabControl1.RemoveAll();
}
```

单击属性窗体里的 TabPages 属性右侧的省略号, 将打开“TabPage 集合编辑器”对话框, 如图 13.33 所示。可以在此窗体中对 TabControl 控件中的选项页进行添加和移除, 并在右侧的属性窗体对相应的选项页的属性进行设置。



图 13.33 “TabPage 集合编辑器”对话框

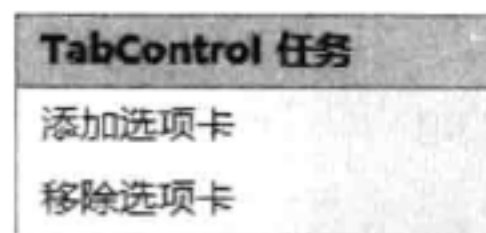


图 13.32 TabControl 任务


TabPage 的 Text 属性指定了标签页的标签头内容,可以在设计时通过属性窗体对 TabPage 的 Text 属性进行设置,也可以在代码中对其进行设置。使用代码设置的方法如下:

```
private void setTabPageText()
{
    //设置标签页标签头文本
    this.tabPage1.Text = "testTabPage";
}
```

如果将 TabControl 控件中的 ShowToolTips 属性设置为 true 时,当鼠标光标在标签头上停留时,将显示此标签页的“工具提示”,对于提示内容可以在每个 TabPage 的 ToolTipText 属性进行设置。

可以在设计时通过属性窗体对 TabPage 的 ToolTipText 属性进行设置,也可以在代码中对其进行设置。使用代码设置的方法如下:

```
private void setTabPageText()
{
    //设置标签页的提示文本
    this.tabPage1.ToolTipText = "这是 tabPage1 的 ToolTip";
}
```

 **技巧:** 通过对 ToolTipText 属性进行设置,可以添加此标签页的帮助说明信息。

运行结果如图 13.34 所示。

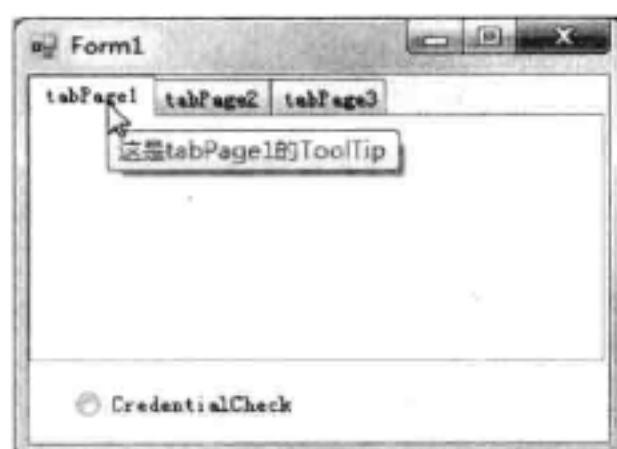


图 13.34 TabPage 的 ToolTip

13.3.7 网格形式容器控件 TableLayoutPanel

TableLayoutPanel 控件在 .NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中,TableLayoutPanel 控件以网格的方式承载并排列其子控件。任何 Windows 窗体控件都可以成为 TableLayoutPanel 控件的子控件,但是 TableLayoutPanel 控件的每个网格中最多只可承载一个子控件。

TableLayoutPanel 控件在 Visual Studio 的工具箱中的图示如图 13.35 所示,将 TableLayoutPanel 控件拖曳到窗体并选定后,如图 13.36 所示。然后,可在 Visual Studio 的属性窗体中对 TableLayoutPanel 控件的属性进行设置,使其达到应用程序的具体需求。



图 13.35 TableLayoutPanel 控件

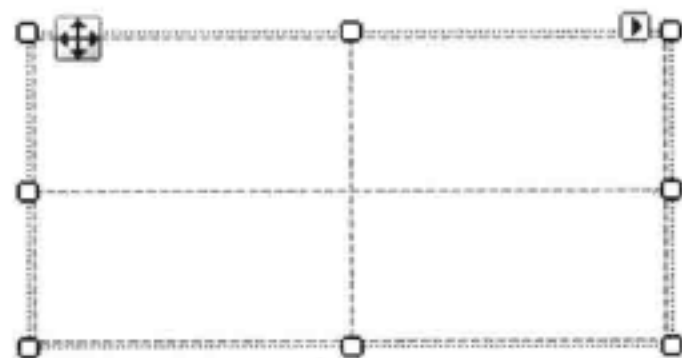


图 13.36 TableLayoutPanel 控件

TableLayoutPanel 控件作为容器控件的一种，可以在其中包含其他的控件。通过 TableLayout Panel 控件，可以将 Windows 窗体进行进一步地细分。

单击 TableLayoutPanel 控件右上角的向右箭头，可以打开“TableLayoutPanel 任务”对话框，如图 13.37 所示。通过单击此对话框中的相应标签，可以对 TableLayoutPanel 控件执行“添加列”、“添加行”、“移除最后一列”、“移除最后一行”和“编辑行和列”等操作。



图 13.37 “TableLayoutPanel 任务”对话框

13.3.8 设置 TableLayoutPanel 控件的外观

单击“TableLayoutPanel 任务”对话框中的“编辑行和列...”标签项，将打开“列和行样式设计”对话框，如图 13.38 所示。在此对话框中，将显示 TableLayoutPanel 控件中所有行和列的基本信息。在显示下拉框中选择想要显示的信息内容，通常为“行”和“列”两项。无论显示行还是列，在下面的 ListView 控件中，都将显示每个列成员或行成员的名称、大小类型和具体值。还可以在对话框右侧的“大小类型”GroupBox 控件中选择希望决定的列或行的大小决定方式。

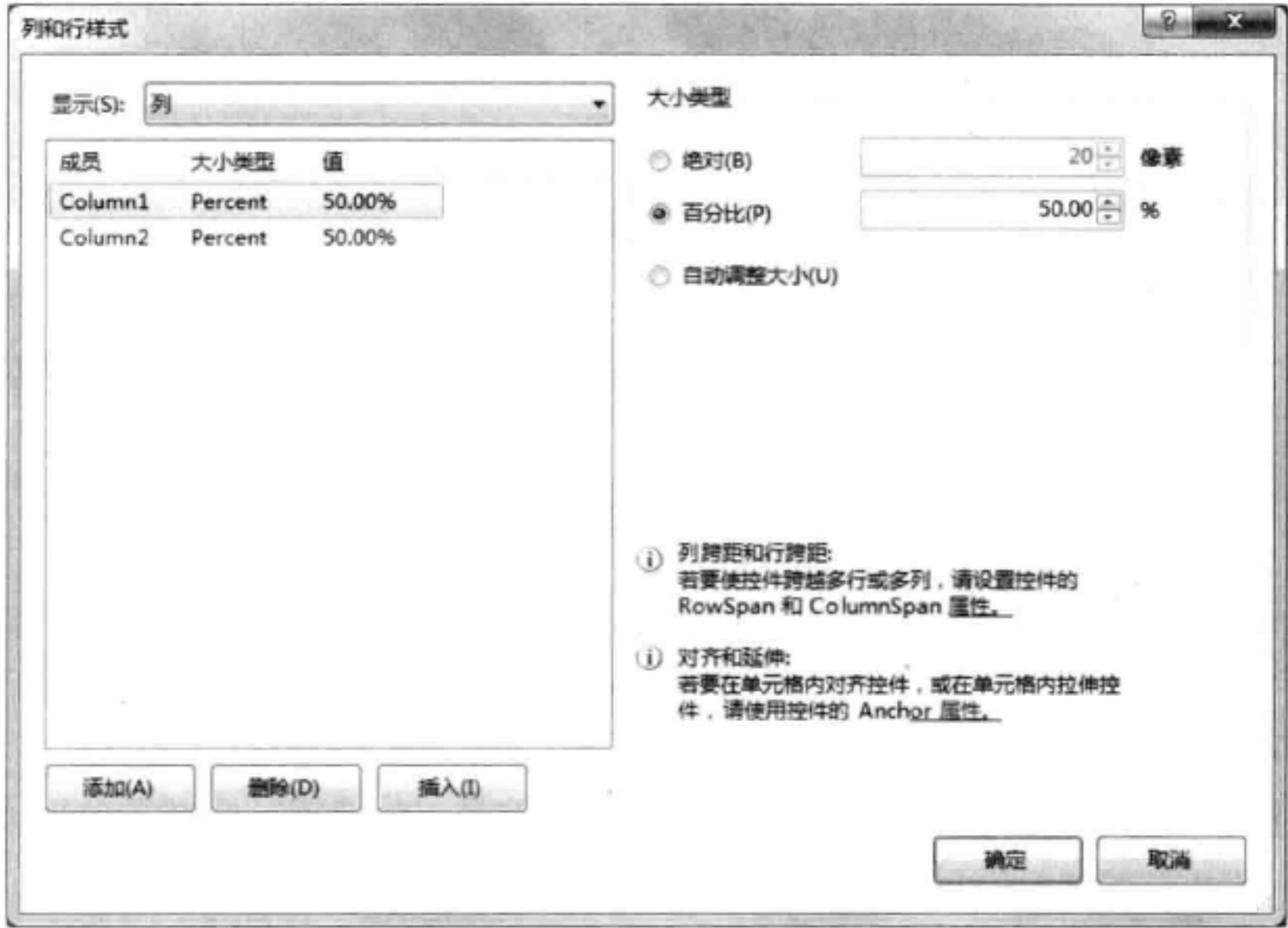


图 13.38 “列和行样式设计”对话框


TableLayoutPanel 控件的 CellBorderStyle 属性用来指示 TableLayoutPanel 控件的单元格的边框样式，它是一个 TableLayoutPanelCellBorderStyle 枚举类型值。TableLayoutPanelCellBorderStyle 的具体成员列表如表 13.11 所示。

表 13.11 TableLayoutPanelCellStyle 成员列表

成 员	说 明	成 员	说 明
None	没有边框	InsetDouble	凹陷的粗边框
Single	细直线边框	OutsetDouble	凸起的粗边框
Inset	凹陷的细边框	OutsetPartial	部分突起的细边框
Outset	凸起的细边框		

可以在设计时通过属性窗体对 TableLayoutPanel 控件的 CellBorderStyle 属性进行设置,也可以在代码中对其进行设置。使用代码设置的方法如下:

```
private void setCellStyle()
{
    this.tableLayoutPanel1.CellBorderStyle =
        TableLayoutPanelCellStyle.OutsetPartial;
}
```

说明: TableLayoutPanel 控件的 ColumnStyles 属性是一个 ColumnStyle 的集合,它包含了 TableLayoutPanel 控件中的每个列的 ColumnStyle。

ColumnStyles 属性是一个 TableLayoutColumnStyleCollection 类的对象,可以使用 TableLayoutColumnStyleCollection 类的 Add()和 Remove()方法向 TableLayoutPanel 控件的 ColumnStyles 属性中添加和移除 ColumnStyle。具体代码如下所示。

☐ 添加 ColumnStyle: 使用 TableLayoutColumnStyleCollection 类的 Add()方法。

```
private void addColumnStyle ()
{
    ColumnStyle myColumnStyle =
        new ColumnStyle(System.Windows.Forms.SizeType.Percent, 50F);
    this.tableLayoutPanel1.ColumnStyles.Add(myColumnStyle );
}
```


☐ 移除 ColumnStyle: 使用 TableLayoutColumnStyleCollection 类的 Remove()方法。

```
private void removeColumnStyle()
{
    ColumnStyle myColumnStyle =
        new ColumnStyle(System.Windows.Forms.SizeType.Percent, 50F);
    this.tableLayoutPanel1.ColumnStyles.Remove(myColumnStyle );
}
```

☐ 清除 ColumnStyle: 若要移除所有 ColumnStyle, 可使用 TableLayoutColumnStyleCollection 类的 clear()方法。

```
private void clearColumnStyle()
{
    this.tableLayoutPanel1.ColumnStyles.clear(tabControl1.SelectedTab);
}
```

ColumnStyle 属性主要用来提供 TableLayoutPanel 控件的每一列的外观设置,一共有两个属性,分别是 Width 属性和 SizeType 属性。

 **说明:** TableLayoutPanel 控件的 RowStyles 属性的使用和设置方法与 ColumnStyle 属性类似。它是一个 RowStyle 的集合,它包含了 TableLayoutPanel 控件中的每个行的 RowStyle。

RowStyles 属性是一个 TableLayoutRowStyleCollection 类的对象,可以使用 TableLayoutRowStyleCollection 类的 Add()和 Remove()方法向 TableLayoutPanel 控件的 RowStyles 属性中添加和移除 RowStyle。

ColumnCount 属性可以获得 TableLayoutPanel 控件中列的个数,RowCount 属性可以获得 TableLayoutPanel 控件中行的个数。在 TableLayoutPanel 控件中,下列表达式恒成立:

```
this.tableLayoutPanel1.ColumnStyles.Count==
    this.tableLayoutPanel1.ColumnCount;
this.tableLayoutPanel1.RowStyles.Count==this.tableLayoutPanel1.RowCount;
```

TableLayoutPanel 控件的 GrowStyle 属性设置了当 TableLayoutPanel 控件中的所有单元格中都含有控件时,若再向 TableLayoutPanel 控件添加子控件,该如何向 TableLayoutPanel 控件添加新的单元格。

GrowStyle 属性值是一个 TableLayoutPanelGrowStyle 枚举类型值。TableLayoutPanelGrowStyle 的具体成员列表如表 13.12 所示。

表 13.12 TableLayoutPanelGrowStyle 成员列表

成 员	说 明
AddColumns	当 TableLayoutPanel 控件中的所有单元格中都含有控件时,若再向 TableLayoutPanel 控件添加子控件,则 TableLayoutPanel 控件将添加新的列
AddRows	当 TableLayoutPanel 控件中的所有单元格中都含有控件时,若再向 TableLayoutPanel 控件添加子控件,则 TableLayoutPanel 控件将添加新的行
FixedSize	当 TableLayoutPanel 控件中的所有单元格中都含有控件时,不能再向 TableLayoutPanel 控件添加子控件

可以在设计时通过属性窗体对 TableLayoutPanel 控件的 GrowStyle 属性进行设置,也可以在代码中对其进行设置。使用代码设置的方法如下:


```
private void setGrowStyle()
{
    this.tableLayoutPanel1.GrowStyle =
        System.Windows.Forms.TableLayoutPanelGrowStyle.AddColumns;
}
```

13.3.9 添加控件到 TableLayoutPanel 控件

当将一个控件添加到 TableLayoutPanel 控件中时,在此控件的属性中将会自动添加 Cell、Column、Row、ColumnSpan 和 RowSpan 共 5 个属性。这 5 个属性可以在该控件的属性窗体里进行设置,也可以使用相应的方法进行设置。

Cell 属性是一对 int 类型的数组,分别代表控件所在的单元格的行索引号和列索引号。具体设置方法如下:

```
private void addControl()
{
    //this.button1 表示了要向 TableLayoutPanel 控件添加的子控件
    //0 代表控件左上角所在的单元格的行索引号
    //1 代表控件左上角所在的单元格的列索引号
    this.tableLayoutPanel1.Controls.Add(this.button1, 0, 1);
}
```

 注意：如果控件跨越多个行或列，则此 Cell 属性值由控件的左上角所处的单元格的行、列索引号来确定。


Column 属性是一个 int 类型的数字，它代表子控件所在的单元格的列索引号。可以在属性窗体中对此属性进行设置，也可以使用 GetColumn()方法和 SetColumn()方法对 Column 属性进行获取和设置，具体代码如下所示。

❑ 获取子控件的 Column 属性：

```
private int getControlColumn()
{
    int columnIndex = 0;
    //this.button1 表示了要获取 Column 属性的子控件
    columnIndex = this.tableLayoutPanel1.GetColumn(this.button1);
    return columnIndex ;
}
```

❑ 设置子控件的 Column 属性：

```
private void setControlColumn()
{
    //this.button1 表示了要设置 Column 属性的子控件
    //0 代表控件左上角所在的单元格的列索引号
    this.tableLayoutPanel1.SetColumn(this.button1, 0);
}
```

 注意：如果子控件跨越多个行，则 Column 属性值由控件的左上角所处的单元格的行索引号来确定。

Row 属性代表了子控件的左上角所在的单元格的行索引号，它的获取与设置方法与 Column 属性非常类似，在此不再赘述。

有时，用户希望可以让一个子控件跨越 TableLayoutPanel 控件的多行或多列，这个时候，可以通过设置子控件的 ColumnSpan 属性和 RowSpan 属性获得。

ColumnSpan 属性是一个 int 类型的数字，它表示子控件在 TableLayoutPanel 控件中跨越的列数。可以在属性窗体中对此属性进行设置，也可以使用 GetColumnSpan()方法和 SetColumnSpan()方法对 ColumnSpan 属性进行获取和设置，具体代码如下所示。

❑ 获取子控件的 ColumnSpan 属性：

```
private int getControlColumnSpan()
{
    int columnSpan = 0;
    //this.button1 表示了要获取 ColumnSpan 属性的子控件
    columnSpan = this.tableLayoutPanel1.GetColumnSpan(this.button1);
}
```



```
return columnSpan ;
}
```

□ 设置子控件的 ColumnSpan 属性:


```
private void setControlColumnSpan()
{
    //this.button1 表示了要设置 ColumnSpan 属性的子控件
    //2 代表子控件跨越的列数
    this.tableLayoutPanel1.SetColumn(this.button1, 2);
}
```

13.4 值设置控件

值设置控件就是一种能够对值进行设置的控件,在.NET 中常用的值设置控件主要有 CheckBox 控件和 RadioButton 控件两种。在下面的各节中,将对它们进行具体介绍。

13.4.1 用 CheckBox 控件选择打开或关闭状态

CheckBox 控件在.NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中,用户可以通过 CheckBox 控件指示某个特定条件是处于打开状态还是处于关闭状态。它常用于为用户提供是/否或真/假选项。因为 CheckBox 彼此独立工作,所以用户可以同时选择任意多个 CheckBox。

 **说明:** 可以通过成组使用 CheckBox 控件以显示多重选项,用户可以从中选择一项或多项。

CheckBox 控件在 Visual Studio 的工具箱中的图示如图 13.39 所示,将 CheckBox 控件拖曳到窗体选定后,如图 13.40 所示。然后,可在 Visual Studio 的属性窗体中对 CheckBox 的属性进行设置,使其达到应用程序的具体需求。

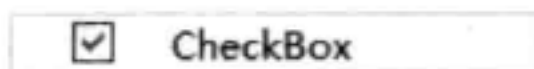


图 13.39 CheckBox 控件

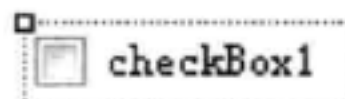


图 13.40 CheckBox 控件

Appearance 属性可以确定 CheckBox 控件的外观,Appearance 属性值是一个 Appearance 枚举类型值。Appearance 的具体成员列表如表 13.13 所示。

表 13.13 Appearance 成员列表

成 员	说 明	成 员	说 明
Button	Button 控件的外观	Normal	系统默认外观,如图 13.40 所示

可以在设计时通过属性窗体对 CheckBox 控件的 Appearance 属性进行设置,也可以在代码中对其进行设置。使用代码设置的方法如下:

```
private void setCheckBoxAppearance()
{
    this.checkBox1.Appearance = System.Windows.Forms.Appearance.Button;
}
```

Checked 属性是一个用于指示当前的 CheckBox 是否选中的 boolean 类型值属性。当 Checked 属性值为 true 时, CheckBox 控件是选中状态, 在 CheckBox 控件中的小方块将会被选中。当 Checked 属性值为 false 时, CheckBox 控件是未选中状态, 在 CheckBox 控件中的小方块中对勾将被取消。

在程序中可以通过对 Checked 属性值的判断来决定程序的运行流程, 具体代码如下:

```
if(this.checkBox1.checked)
{
    ...
}
else
{
    ...
}
```

ThreeState 属性是一个用于指示当前的 CheckBox 是否允许 3 种复选状态的 boolean 类型值属性。当 ThreeState 属性值为 true 时, CheckBox 控件的状态不再是由 Checked 属性标志, 而是由 CheckState 属性标志出来。当 ThreeState 属性值为 false 时, CheckBox 控件还是只有两种状态, 即选中状态和未选中状态。可以在设计时通过属性窗体对 CheckBox 控件的 ThreeState 属性进行设置, 也可以在代码中对其进行设置。使用代码设置的方法如下所示:

```
private void setCheckBoxThreeState()
{
    this.checkBox1.ThreeState=true;
}
```

说明: CheckState 属性在 ThreeState 属性值为 true 时有意义。

CheckState 属性值是一个 CheckState 枚举类型的值。CheckState 的具体成员列表如表 13.14 所示。

表 13.14 CheckState 成员列表

成 员	说 明	成 员	说 明
Checked	CheckBox 控件为选中状态	Indeterminate	CheckBox 控件为不确定状态
Unchecked	CheckBox 控件为未选中状态		

当 CheckState 属性值为 Checked 时, 在 CheckBox 控件中的小方块将会被选中。当 Checked 属性值为 Unchecked 时, 在 CheckBox 控件中的小方块中的对勾将被取消。而当 CheckState 属性值为 Indeterminate 时, 在 CheckBox 控件中的小方块将会被灰色填充。

在程序中可以通过对 CheckState 属性值的判断来决定程序的运行流程, 具体代码如下:

```
if(this.checkBox1.CheckState==CheckState.Checked)
{
```



```

...
}
else
{
    if (this.checkBox1.CheckState==CheckState.Unchecked)
    {
        ...
    }
    else
    {
        ...
    }
}
}

```

AutoCheck 属性是一个 boolean 类型值的变量，当 AutoCheck 属性值为 true，并且 CheckBox 控件被单击时，CheckBox 控件会自动更改 Checked 属性和 CheckState 属性。对于 AutoCheck 属性可以在设计时通过属性窗体对其进行设置，也可以在代码中对其进行设置。使用代码设置的方法如下：

```

private void setCheckBoxAutoCheck()
{
    this.checkBox1.AutoCheck= true;
}

```

如果 AutoCheck 属性值为 false，则可以在 click 事件中编写程序处理代码对程序的 Checked 属性或 CheckState 属性进行设置。具体代码如下所示。

❑ 对 Checked 属性进行设置：

```

private void checkBox1_Click(object sender, System.EventArgs e)
{
    if (checkBox1.Checked)
    {
        //如果复选框被选中，则复选框文本变为"Checked"
        checkBox1.Text = "Checked";
    }
    else
    {
        //如果复选框未被选中，则复选框文本变为" Unchecked "
        checkBox1.Text = "Unchecked";
    }
}

```

❑ 对 CheckState 属性进行设置：

```

private void checkBox1_Click(object sender, System.EventArgs e)
{
    switch (checkBox1.CheckState)
    {
        case CheckState.Checked:
            break;
        case CheckState.Unchecked:
            break;
    }
}

```


```

        case CheckState.Indeterminate:
            break;
    }
}

```

13.4.2 用 RadioButton 控件进行多选一操作

RadioButton 控件在 .NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中，RadioButton 控件为用户提供由两个或多个互斥选项组成的选项集。

 **说明：**复选框 (CheckBox) 控件和单选按钮 (RadioButton) 控件的相似之处在于，它们都是用于指示用户所选的选项。不同之处在于，对于复选框 (CheckBox) 控件，可以选择任意数量的复选框，但是在一个单选按钮组中一次只能选择一个单选按钮。

RadioButton 控件在 Visual Studio 的工具箱中的图示如图 13.41 所示，将 RadioButton 控件拖曳到窗体选定后，如图 13.42 所示。下一步，可在 Visual Studio 的属性窗体中对 RadioButton 控件的属性进行设置，使其达到应用程序的具体需求。



图 13.41 RadioButton 控件

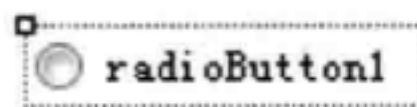


图 13.42 RadioButton 控件

Appearance 属性可以确定 RadioButton 控件的外观，Appearance 属性值是一个 Appearance 枚举类型值，可以在设计时通过属性窗体对 CheckBox 控件的 Appearance 属性进行设置，也可以在代码中对其进行设置。使用代码设置的方法如下：

```

private void setRadioButtonAppearance()
{
    //设置单选框按钮外观
    this.radioButton1.Appearance = System.Windows.Forms.Appearance.Button;
}

```

Checked 属性是一个用于指示当前的 RadioButton 是否选中的 boolean 类型值属性。当 Checked 属性值为 true 时，RadioButton 控件是选中状态，在 RadioButton 控件中的小圆圈将会以绿色填充。当 Checked 属性值为 false 时，RadioButton 控件是未选中状态，在 RadioButton 控件中填充的小圆圈将会取消。

在程序中，可以通过对一组 RadioButton 的 Checked 属性值的判断来决定程序的运行流程，具体代码如下：

```

if(this.radioButton1.checked)
{
    ...
}
if(this.radioButton2.checked)
{
    ...
}

```


AutoCheck 属性是一个 boolean 类型值的变量，当 AutoCheck 属性值为 true，且 RadioButton 控件被单击时，RadioButton 控件会自动更改 Checked 属性。对于 AutoCheck 属性可以在设计时通过属性窗体对其进行设置，也可以在代码中对其进行设置，使用代码设置的方法如下：

```
private void setCheckBoxAutoCheck()
{
    this.checkBox1.AutoCheck= true;
}
```

如果 AutoCheck 属性值为 false，则可以在 click 事件中编写程序处理代码对程序的 Checked 属性进行设置。具体代码如下：

```
private void radioButton1_Click(object sender, System.EventArgs e)
{
    if (radioButton1.Checked)
    {
        //如果单选框被选中，则单选框文本变为"Checked"
        radioButton1.Text = "Checked";
    }
    else
    {
        //如果单选框未被选中，则单选框文本变为"Unchecked"
        radioButton2.Text = "Unchecked";
    }
}
```

13.5 本章总结

在本章中，主要从控件的常用属性、方法和事件 3 个方面，向读者介绍了文本编辑控件、文本显示控件、容器控件和值设置控件 4 大类型的控件。读者通过本章与后面几章的学习，可以更好地了解 Visual Studio 2010 的控件结构，熟练地掌握 Windows 应用程序设计与开发的技巧。

13.6 实战练习

1. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，将 Form1 窗体设计为一个登录窗体，要求用户输入用户名和密码，单击“登录”按钮时进行用户名和密码的验证，若用户名和密码都为 admin，就显示欢迎信息，否则显示出错提示信息。

2. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在 Form1 窗体中设计一个用户登记界面，要求输入姓名、性别、年龄、文化程度、联系电话、手机号码、住址、备注等内容。在设计窗体界面时，要求用 TabControl 控件将用户信息分为两页。

3. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在 Form1 窗体中通过 TableLayoutPanel 控件模拟设计 Windows 8 的 Metro 界面。

第 14 章 列表选择控件介绍

在 Windows 应用程序的开发中用到的列表选择控件主要有列表框、复选列表框、下拉列表框、列表视图、树视图等几种。本章将在各节中对它们进行详细的介绍。

14.1 ListBox 控件

ListBox 控件显示一个项列表，用户可以从中选择一项或多项。如果项总数超出可以显示的项数，则自动向 ListBox 控件添加滚动条。

14.1.1 用 ListBox 显示列表

ListBox 控件在 .NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中，ListBox 控件用于显示一个项列表，用户可从中选择一项或多项。ListBox 控件在 Visual Studio 的工具箱中的图示如图 14.1 所示，将 ListBox 控件拖曳到窗体选定后，如图 14.2 所示。下一步，可在 Visual Studio 的属性窗体中对 ListBox 的属性进行设置，使其达到应用程序的具体需求。



图 14.1 ListBox 控件

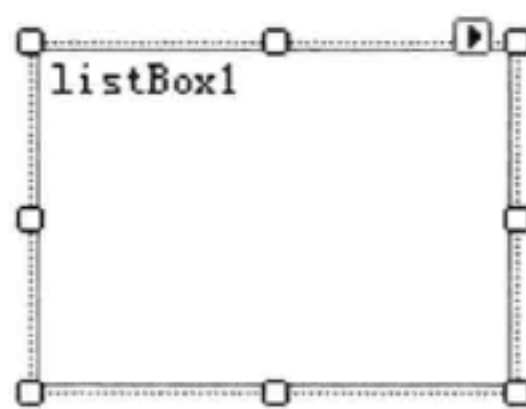


图 14.2 ListBox 控件

14.1.2 用 ListBox 添加列表项

在 ListBox 控件中，有一个最重要的属性，即 Items 属性，此属性是一个 ObjectCollection 类型的实例，用来表示 ListBox 控件中的各项的集合。Items 属性可以在属性窗体中通过打开“字符串集合编辑器”进行编辑，也可以通过使用 ObjectCollection 类的方法和属性进行读取和设置，具体方法可分为以下几种。

- 使用 AddRange() 方法向 Items 属性添加项。

```
private void SetListBoxItems()
```



```

{
    ListBox listBox ;
    listBox= new ListBox();
    //向 ListBox 控件添加子项
    listBox.Items.AddRange(new string[] { "1000.", "1500.", "2000." });
}

```

❑ 使用 Add()方法向 Items 属性添加项。

```

private void SetListBoxItems()
{
    ListBox listBox ;
    listBox= new ListBox();
    //向 ListBox 控件添加子项
    listBox.Items.Add( "1000.");
    listBox.Items.Add( "1500.");
    listBox.Items.Add( "2000.");
}

```

❑ 使用 Insert()方法向 Items 属性的指定位置添加项。

```

private void SetListBoxItems()
{
    ListBox listBox ;
    listBox= new ListBox();
    //向 ListBox 控件添加子项
    listBox.Items.Insert ( "1000.");
    listBox.Items.Insert ( "2000.");
    listBox.Items.Insert (1, "1500.");
}

```

这段代码的运行结果与前面两个示例相同。

❑ 使用 Remove()方法去除 Items 属性中的某一项。

```

private void RemoveListBoxItems()
{
    ListBox listBox ;
    listBox= new ListBox();
    //向 ListBox 控件添加子项
    listBox.Items.Add( "1000.");
    listBox.Items.Add( "1500.");
    listBox.Items.Add( "2000.");
    //向 ListBox 控件移除子项
    listBox.Items.Remove( "1500.");
}

```

❑ 使用 RemoveAt()方法删除 Items 属性中的某一项。

```

private void RemoveListBoxItems()
{
    ListBox listBox ;
    listBox= new ListBox();
    listBox.Items.Add( "1000." );
    listBox.Items.Add( "1500.");
}

```

```

listBox.Items.Add( "2000.");
//向 ListBox 控件移除子项
listBox.Items.RemoveAt(1);
}

```

❑ 使用 Clear()方法可以删除 Items 属性中的所有项。

```

private void ClearListBoxItems()
{
    ListBox listBox ;
    listBox= new ListBox();
    listBox.Items.Add( "1000." );
    listBox.Items.Add( "1500.");
    listBox.Items.Add( "2000.");
    //清除 ListBox 控件中各子项
    listBox.Items.Clear();
}

```

⚠注意: Items 属性是一个 ObjectCollection 类型的实例。

在使用 Add()方法后, ListBox 控件的外观如图 14.3 所示。

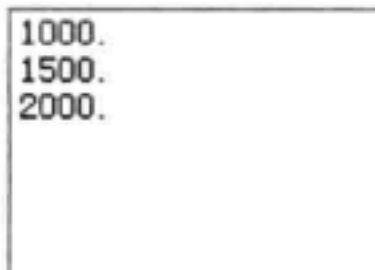


图 14.3 ListBox 控件

14.1.3 设置 ListBox 的行为

IntegralHeight 属性是一个 boolean 类型的值,它用来指示 ListBox 控件是否自适应的调整大小,使 ListBox 控件中显示出来的项完全显示。当 IntegralHeight 属性值为 true 时,ListBox 控件中显示出来的项不会只显示一部分。可以在属性窗体中对 IntegralHeight 属性进行设置,也可以使用以下代码对其进行设置:

```

private void SetListBoxIntegralHeight()
{
    ListBox listBox ;
    listBox= new ListBox();
    listBox.IntegralHeight=true;
}

```

ItemHeight 属性是一个 int 类型的值,它代表了 ListBox 控件中各项的高度,可以在属性窗体中对 ItemHeight 属性进行设置,也可以使用以下代码对其进行设置:

```

private void SetListBoxIntegralHeight()
{
    ListBox listBox ;
    listBox= new ListBox();
    listBox.ItemHeight=14;
}

```

ListBox 控件中各项的宽度,可以使用下面的表达式获得:

```

(int)listBox.CreateGraphics().MeasureString(listBox.Items[index].ToString(),listBox.Font).Width;

```

PreferredHeight 属性也是一个 int 类型的值,它表示的是 ListBox 控件需要设定的高度,

而这个高度可以使 ListBox 控件中的所有项都完全显示。

注意： PreferredHeight 属性是对应于 ListBox 控件中所有项都可显示，而 IntegralHeight 属性则是对应于 ListBox 控件中显示在显示屏上的项。

可以通过将 PreferredHeight 属性设置给 ListBox 控件的 Height 属性，而使 ListBox 控件中的所有项都可以完全显示，而不需要自动添加滚动条。具体的使用代码如下所示：

```
private void SetlistBoxHeight()
{
    //向 ListBox 控件添加项。
    for(int x = 0; x < 20; x++)
    {
        listBox.Items.Add("第" + x.ToString()+"项");
    }
    //设置 ListBox 控件的高度，使其中的项可以完全显示出来
    listBox.Height = listBox.PreferredHeight;
}
```

MultiColumn 属性是一个 boolean 类型的变量，它用于指示当 ListBox 控件中的各项的高度之和超过了 ListBox 控件的高度时，是否可以多列显示。对 MultiColumn 的设置可以使用下面的代码：

```
private void SetlistBoxMultiColumn()
{
    this.listBox.MultiColumn = true;
}
```

当 ListBox 控件的 MultiColumn 属性为 false 时，ListBox 控件的显示方式如图 14.4 所示。而当 ListBox 控件的 MultiColumn 属性为 true 时，ListBox 控件的显示方式如图 14.5 所示。



图 14.4 ListBox 控件

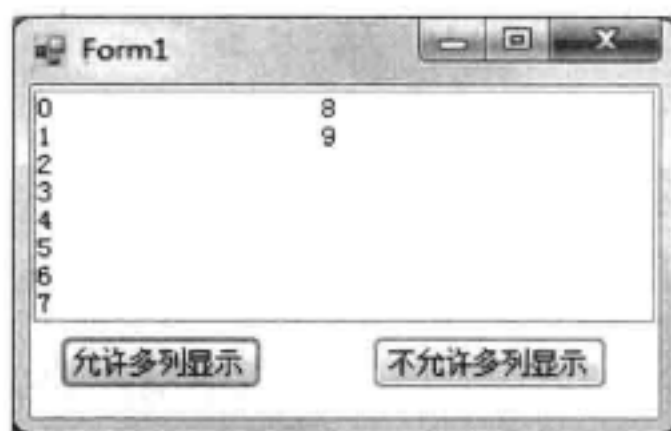


图 14.5 ListBox 控件

如图 14.6 所示，当 ListBox 控件的 MultiColumn 属性为 true 时，如果 ListBox 中各列的宽度过宽，则会产生水平滚动条。这时可以通过 ColumnWidth 属性进行设置，来控制各列的宽度。

在下面的代码中，程序获得 ListBox 中最后一项的宽度并将它赋值给 ColumnWidth 属性。具体代码如下所示：

```
private void btnGO_Click(object sender, EventArgs e)
{
    this.listBox.ColumnWidth = this.listBox.Items[this.listBox.Items.Count - 1].Text.Length * 8;
}
```

```
//向 ListBox 控件添加子项
this.listBox.Items.Clear();
this.listBox.Items.Add("0");
this.listBox.Items.Add("1");
this.listBox.Items.Add("2");
this.listBox.Items.Add("3");
this.listBox.Items.Add("4");
this.listBox.Items.Add("5");
this.listBox.Items.Add("6");
this.listBox.Items.Add("7");
this.listBox.Items.Add("8");
this.listBox.Items.Add("9");
int index = listBox.Items.Count - 1;
//获得 ListBox 中最后一项的宽度并将它赋值给 ColumnWidth 属性
string content=listBox.Items[index].ToString();
int width = (int)listBox.CreateGraphics().MeasureString(content,
    listBox.Font).Width;
listBox.ColumnWidth = width;
}
```

运行结果如图 14.7 所示。

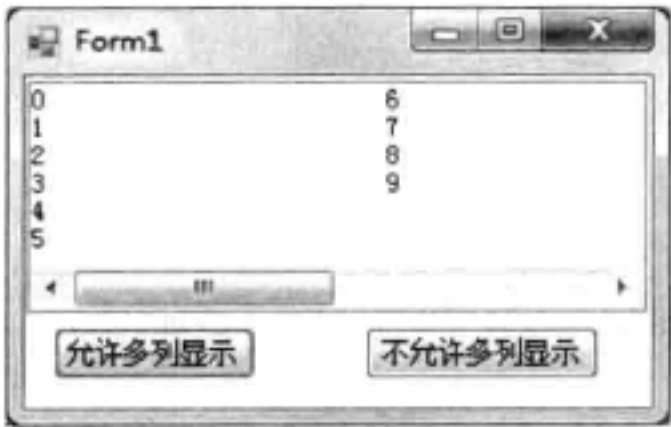


图 14.6 ListBox 控件



图 14.7 ListBox 控件

14.1.4 让 ListBox 支持多选

ListBox 控件的 SelectionMode 属性是一个 SelectionMode 枚举类型的值，它用来表示 ListBox 控件所确定的选择模式。SelectionMode 的具体成员列表如表 14.1 所示。

表 14.1 SelectionMode成员列表

成 员	说 明
None	ListBox 控件中无法进行选择
One	ListBox 控件中最多可以选择一项
MultiSimple	ListBox 控件中可以进行多项选择
MultiExtended	ListBox 控件中可以进行多项选择，且支持键盘上 Shift 键、Ctrl 键和方向键的使用

在下面的代码中，以编程的方式设置了 ListBox 控件的 SelectionMode。代码设置如下所示。

```
private void SetListBoxSelectionMode()
{
    ListBox listBox ;
```




```

listBox= new ListBox();
//设置 ListBox 控件的选择模式
listBox.SelectionMode=SelectionMode.MultiSimple;
}

```

在上面的代码中，将 ListBox 控件的选择模式设置为可进行多项选择。

说明：当 ListBox 控件的 SelectionMode 属性设置为 One 时，可使用 ListBox 控件的 SelectedItem 属性获得用户当前选中的项，或使用 SelectedIndex 属性获得用户当前选中项的索引号。选中项的文本内容可使用 ListBox 控件的 Text 属性获得。

在下面的代码中，当用户单击窗体下方的按钮时，如图 14.8 所示，将弹出一个消息框，提示用户当前选定的项的索引号及其对应的文本内容，如图 14.9 所示。代码设置如下：

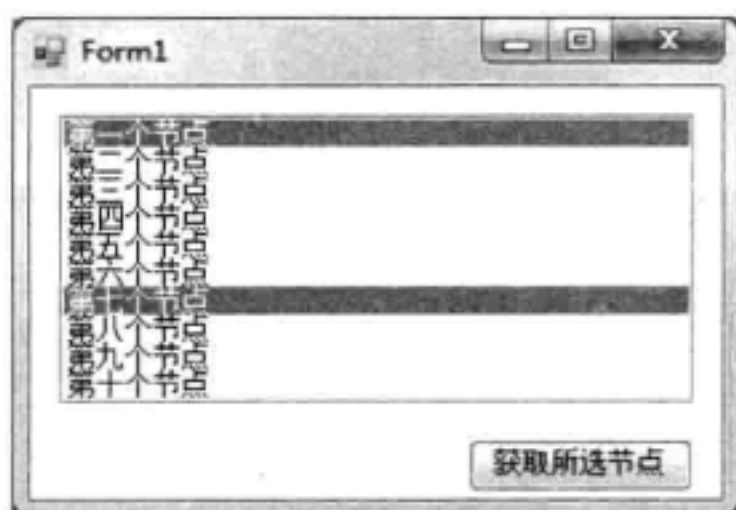


图 14.8 ListBox 控件

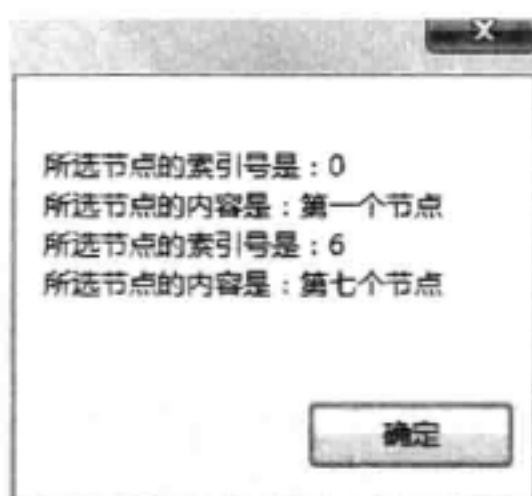


图 14.9 消息提示框

```

public _InitializeComponent()
{
    //设置 ListBox 控件的选择模式
    this.listBox1.SelectionMode = System.Windows.Forms.SelectionMode.
    MultiExtended;
    //向 ListBox 控件添加子项
    this.listBox1.Items.Add("第一个节点");
    this.listBox1.Items.Add("第二个节点");
    this.listBox1.Items.Add("第三个节点");
    this.listBox1.Items.Add("第四个节点");
    this.listBox1.Items.Add("第五个节点");
    this.listBox1.Items.Add("第六个节点");
    this.listBox1.Items.Add("第七个节点");
    this.listBox1.Items.Add("第八个节点");
    this.listBox1.Items.Add("第九个节点");
    this.listBox1.Items.Add("第十个节点");
}

```

为窗体里的 Button 按键添加事件处理函数。代码设置如下所示。

```

private void button1_Click(object sender, EventArgs e)
{
    string message = "";
    for (int i = 0; i < this.listBox1.SelectedItems.Count; i++)
    {
        message = message + "所选节点的索引号是：" +
            this.listBox1.SelectedIndices[i] + "\n" + "所选节点的内容是："
    }
}

```


```

        + this.listBox1.SelectedItems[i].ToString()+"\n";
    }
    //显示消息对话框，提示所选节点的索引号和内容
    MessageBox.Show(message);
}

```

14.1.5 设置 ListBox 的外观

ListBox 控件的 HorizontalScrollbar 属性和 ScrollAlwaysVisible 属性分别用来表示在 ListBox 控件中是否要显示水平滚动条和水平滚动条是否永远显示。这两个属性都是 boolean 类型的值。

说明：当 HorizontalScrollbar 属性为 true 时，可以使用 HorizontalExtent 属性来定义或获得水平滚动条可滚动的宽度。

可以在属性窗体中对这 3 个属性进行设置，也可以使用以下代码对其进行设置：

```

private void SetListBoxScrollbar()
{
    ListBox listBox ;
    listBox= new ListBox();
    //设置 ListBox 控件的滚动条
    listBox.HorizontalScrollbar=true;
    listBox.ScrollAlwaysVisible=true;
    //设置水平滚动条可滚动的宽度
    listBox.HorizontalExtent=this.listBox1.ColumnWidth*5;
}

```

在上面这段代码中，将 ListBox 控件的水平滚动条设置为可以显示，将 ListBox 控件的垂直滚动条设置为永久显示，并将水平滚动条的宽度设置为 ListBox 控件中每一列的宽度的 5 倍，即在此 ListBox 控件中，当可多列显示时，最多可显示 5 列。

ListBox 控件的 TopIndex 属性用来定义和获得 ListBox 控件中的第一个显示出来的项在 Items 属性中的索引值。可以在属性窗体中对 TopIndex 属性进行设置，也可以使用以下代码对其进行设置：

```

private void SetListBoxScrollbar()
{
    ListBox listBox ;
    listBox= new ListBox();
    //设置 ListBox 控件的第一个显示的项
    listBox.TopIndex=3;
}

```

在上面的代码中，将 TopIndex 属性设为 3，即在 ListBox 控件启动时，第一个显示出来的是在 Items 属性里索引号为 3 的项，也就是 ListBox 控件的第 4 项。因此 ListBox 控件的前 3 个项在启动时并不显示，但可以通过滚动 ListBox 控件右侧自动生成的垂直滚动条来浏览。

在下面的代码示例中，可以随时将用户当前选定的项设置为 ListBox 控件中的第一个

显示的项，具体代码如下所示。

```
private void SetListBoxScrollbar()
{
    //设置 ListBox 控件的选择模式
    this.listBox1.SelectionMode = System.Windows.Forms.SelectionMode.One;
    this.listBox1.TopIndex=this.listBox1.SelectedIndex
}
```

14.1.6 对 ListBox 列表进行排序


ListBox 控件的 Sorted 属性用来定义 ListBox 控件中的项是否按字典序排序。可以在属性窗体中对 Sorted 属性进行设置，也可以使用以下代码对其进行设置：

```
private void SetListBoxSorted()
{
    ListBox listBox;
    listBox= new ListBox();
    //设置 ListBox 控件的项是否排序
    listBox.Sorted=true;
}
```

在这段代码中，ListBox 控件中的各项可自动按字典序进行排序。另外，也可以使用 ListBox 控件的 Sort()方法对 ListBox 控件中的项进行排序，如下代码所示。

```
private void button1_Click(object sender, EventArgs e)
{
    //对 ListBox 控件的子项进行排序
    this.listBox1.Sort();
}
```

使用上面这段代码时，每当单击 button1，则 listBox1 中的项将自动排序。

 注意：默认的排序方式是按字典顺序进行排序。

14.1.7 控制 ListBox 列表项的刷新

使用 ListBox 控件的 BeginUpdate()方法可以防止 ListBox 控件的绘图功能，而 EndUpdate()方法则可以重新开始 ListBox 控件的绘图功能。比如，在向 ListBox 控件中添加新的项时，每添加一次 ListBox 控件就会自我绘制一次，如果在程序中向 ListBox 控件添加了大量的项时，则会使应用程序中的 ListBox 控件反复刷新，不但占用资源，也给用户一种程序不稳定的感觉。这个时候，推荐使用 BeginUpdate()方法使 ListBox 控件先停止自我绘制，然后再在控件中添加或删除项，最后，使用 EndUpdate()方法刷新 ListBox 控件。

在下面的代码中，将通过调用 BeginUpdate()方法和 EndUpdate()方法向 ListBox 控件添加项。代码设置如下所示。

```
public void AddItemToListBox()
{
```

```
//停止 ListBox 控件的绘图功能
listBox1.BeginUpdate();
//向 ListBox 控件中添加 1000 个新项
for(int x = 1; x < 1000; x++)
{
    listBox1.Items.Add("项"+x.ToString());
}
//. 启动 ListBox 控件的绘图功能
listBox1.EndUpdate();
}
```

读者通过运行下面的代码，比较两者的区别：

```
public void AddItemToListBox()
{
    //向 ListBox 控件中添加 1000 个新项
    for(int x = 1; x < 1000; x++)
    {
        listBox1.Items.Add("项"+x.ToString());
    }
}
```

14.1.8 查找 ListBox 中的列表

FindString()方法主要是用来查找在 ListBox 控件中以制定字符串相匹配的项。FindString()方法使用了重载机制来完成，共有两种使用它的途径，分别介绍如下。

其一是，在 ListBox 控件中查找以参数中的字符串为开始的项，查找结束后返回所查找项的索引，代码设置如下：

```
ListBox.FindString(string)
```

□ string 类型的参数是指要进行查找的字符串。

在下面的代码中，单击窗体中的查找按钮，将在 ListBox 控件中查找相应的字符串匹配的项，并通过消息框通知用户该字符串所在的项的索引，然后将此项选中。如果没有匹配的项，也会用消息框通知用户，代码设置如下：


```
private void btnFind_Click(object sender, EventArgs e)
{
    //txtFind 是一个文本框
    //用户在此文本框中输入想要查找的信息
    string searchText = this.txtFind.Text ;
    //保证 txtFind 中有内容
    if (searchText != null && searchText != "")
    {
        //在 ListBox 控件中查找是否有匹配的项
        int index = listBox1.FindString(searchText);
        //判断是否查找到相匹配的项
        if (index != -1)
        {
            //如果有匹配的项，则选定此项
        }
    }
}
```



```

        listBox1.SetSelected(index, true);
        //以消息框方式告知用户此项的索引号
        MessageBox.Show("在 ListBox 控件中的第"+index.ToString()+"项具有"+
            searchText+"字符串");
    }
    else
    {
        //没有匹配的项, 则以消息框方式提示
        MessageBox.Show("在 ListBox 控件中没有匹配的项");
    }
}
}

```

 注意: FindString()方法使用了重载机制来完成。

其二是, 在 ListBox 控件中从指定的项开始查找以参数中的字符串为开始的项, 查找结束后返回所查找项的索引代码如下:

```
ListBox.FindString(string, int)
```

- string 类型的参数是指要进行查找的字符串。
- int 类型的参数是代表指定的开始查找的项的索引号。

在下面的代码中, 单击窗体中的查找按钮, 将从当前用户选定的项开始查找用户在文本框中输入的字符串中与相应的字符串匹配的项, 并将此项选中, 通过消息框通知用户该字符串所在的项的索引。如果没有匹配的项, 也会用消息框通知用户。

```


private void btnFind_Click(object sender, EventArgs e)
{
    //txtFind 是一个文本框
    //用户在此文本框中输入想要查找的信息
    string searchText = this.txtFind.Text ;
    //获得当前用户选定的项的索引
    int startIndex = this.listBox1.SelectedIndex;
    //保证 txtFind 中有内容
    if (searchText != null && searchText != "")
    {
        //在 ListBox 控件中查找是否有匹配的项
        int index = listBox1.FindString(searchText, startIndex);
        //判断是否查找到相匹配的项
        if (index != -1)
        {
            //如果有匹配的项, 则选定此项
            listBox1.SetSelected(index, true);
            //以消息框方式告知用户此项的索引号
            MessageBox.Show("在 ListBox 控件中的第"+index.ToString()+"项具有"+
                searchText+"字符串");
        }
        else
        {
            //没有匹配的项, 则以消息框方式提示
            MessageBox.Show("在 ListBox 控件中没有匹配的项");
        }
    }
}

```

```

    }
}
}

```

 **注意：**使用 ListBox 控件的 FindString() 方法查找的匹配项不能区分大小写的匹配。如果需要区分大小写，可以使用 FindStringExact() 方法来实现，具体的使用方法与 FindString() 方法相类似，在此不再赘述。

14.1.9 ListBox 控件的常用方法

ListBox 控件的 GetItemHeight() 方法可用于获得指定项的高度。在下面的代码中，将使用 GetItemHeight() 方法获取 ListBox 控件中各项的高度，并将这些项的高度和设置为 ListBox 控件的高度。

```

private void button1_Click(object sender, EventArgs e)
{
    //初始化变量
    int itemHeight = 0;
    int sumHeight = 0;
    int itemCount = 0;
    //获得 ListBox 控件中项的个数
    itemCount = this.listBox1.Items.Count;
    //求各项的高度和
    for (int i = 0; i < itemCount; i++)
    {
        itemHeight = this.listBox1.GetItemHeight(i);
        sumHeight = sumHeight + itemHeight;
    }
    //将各项的高度和设置为 ListBox 控件的高度
    this.listBox1.Height = sumHeight;
}

```

ListBox 控件的 SetItemsCore() 方法可用于清除控件中的项，并向 ListBox 控件中添加其他项。下面的两段代码所得到的结果相同，均是将 ListBox 控件中的原有项删除并添加新的项，但是实现方法不同，请读者认真体会。

☐ 使用 Items 属性的 Clear() 和 AddRange() 方法，代码设置如下：

```

private void setItems()
{
    //清除 ListBox 控件中的子项
    this.listBox1.Items.Clear();
    //向 ListBox 控件添加项
    this.listBox1.Items.AddRange(new object[] { "第一项", "第二项", "第三项",
        "第四项", "第五项" });
}

```


□ 使用 ListBox 控件的 SetItemsCore()方法，代码设置如下：

```
private void setItems()
{
    //清除 ListBox 控件中的子项
    this.listBox1.Items.Clear();
    //向 ListBox 控件添加项
    this.listBox1.SetItemsCore(new object[] { "第一项", "第二项", "第三项",
        "第四项", "第五项" });
}
```

🔗 技巧：还可以通过 ListBox 控件的方法对 ListBox 控件中的选定项进行操作。

ClearSelected()方法可以取消 ListBox 控件中对任意项的选定，在 ListBox 控件中用过此方法后，ListBox 控件中将没有选定的项。而 GetSelected()方法则返回一个 boolean 类型的变量，用来判断在应用程序中，用户是否选择了参数所指定的项。如果选定了就返回 true，如果没有选定则返回 false。而 SetSelected()方法则用来在程序中设置对指定项的选定与否。SetSelected()方法拥有两个参数，第一个参数是一个 int 类型的值，用来指示操作项的索引。第二个参数是一个 boolean 类型的值，用来指示该项的选中与否，如果此值为 true，则代表选中第一个参数所指定的项。如果此值为 false，则代表不选中第一个参数所指定的项。

在下面的代码中，将对 ListBox 控件中的项进行反选。即对已经选中的项取消选中，而对没有选中的项进行选中。

```
private void button1_Click(object sender, EventArgs e)
{
    int itemCount = 0;
    //获取 ListBox 控件中的子项的个数
    itemCount = this.listBox1.Items.Count;
    for (int i = 0; i < itemCount; i++)
    {
        bool ifSelected = this.listBox1.GetSelected(i);
        //如果 ListBox 控件中的子项被选中，则将此子项设为未选中
        if (ifSelected)
        {
            SetSelected(i, false);
        }
        //如果 ListBox 控件中的子项未选中，则将此子项设为被选中
        else
        {
            SetSelected(i, true);
        }
    }
}
```

14.2 ComboBox 控件

ComboBox（组合框）控件和 ListBox（列表框）控件的功能类似，但使用更为灵活，更常用。实际上，组合框是由一个文本框和一个列表框组成的，单击文本框右侧的下拉按钮即可展开下拉列表。

14.2.1 认识 ComboBox 控件

ComboBox 控件在 .NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中，ComboBox 控件用于在下拉组合框中显示数据。在默认情况下，ComboBox 控件分两个部分显示：顶部是一个允许用户输入列表项的文本框。第二部分是一个列表框，它显示一个项目列表，用户可以从中选择一项。

ComboBox 控件在 Visual Studio 的工具箱中的图示如图 14.10 所示，将 ComboBox 控件拖曳到窗体并选定后，如图 14.11 所示。下一步，可在 Visual Studio 的属性窗体中对 ComboBox 的属性进行设置，使其达到应用程序的具体需求。



图 14.10 ComboBox 控件



图 14.11 ComboBox 控件

14.2.2 设置 ComboBox 下拉样式

DropDownStyle 属性是 ComboBox 控件最重要的一个属性，它用于指定 ComboBox 控件组合框的样式。DropDownStyle 属性值是一个 ComboBoxStyle 枚举类型的值，它用来表示 ComboBox 控件的样式。DropDownStyle 的具体成员列表如表 14.2 所示。

表 14.2 DropDownStyle 成员列表


成 员	说 明
DropDown	用户既可以在列表项中进行选择，也可以在文本框中直接填写
DropDownList	用户只可以在列表项中进行选择
Simple	用户既可以在列表项中进行选择，也可以在文本框中直接填写，并且列表框是一直是可见的

在下面的代码中，以编程的方式设置了 ComboBox 控件的 DropDownStyle。

```
private void SetComboBoxDropDownStyle()
{
    //设置 ComboBox 控件的下拉框样式
    this.comboBox1.DropDownStyle = System.Windows.Forms.ComboBoxStyle.
```



```
DropDownList;
}
```

 **注意：** DropDownStyle 属性是 ComboBox 控件的一个重要属性，只能在代码中进行设定。

对于 ComboBox 控件的下拉列表，还有以下几个重要的属性需要注意。

DroppedDown 属性： 是一个 boolean 类型的属性值，可以用来指示在程序运行时，ComboBox 控件是否显示下拉列表。在编程过程中，可以通过对此属性进行判断来确定 ComboBox 控件所处的状态，也可以通过对此属性进行设置来控制 ComboBox 控件的状态。

DropDownHeight 属性： 是一个 int 类型的属性值，用来指示 ComboBox 控件中下拉列表的高度。当对下拉列表框的高度设置大于下拉列表框中的各项的高度和时，下拉列表框的显示高度为列表框中各项的高度和，如图 14.12 所示。当下拉列表框的高度设置小于下拉列表框中的各项的高度和时，下拉列表的右侧将出现一对带有方向箭头的按钮，可以通过单击这两个按钮，来移动列表中的内容，以便浏览，如图 14.13 所示。

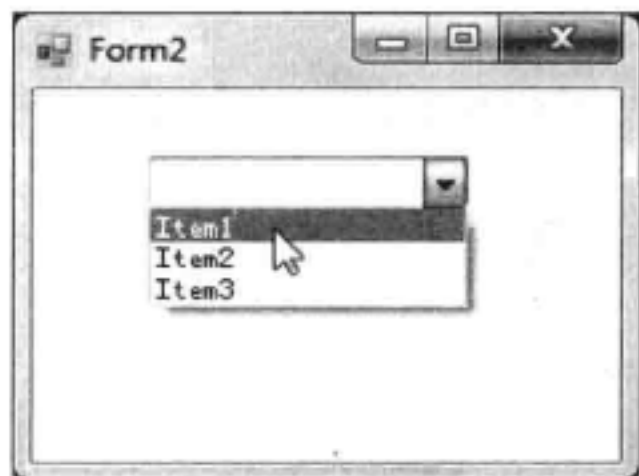


图 14.12 ComboBox 控件

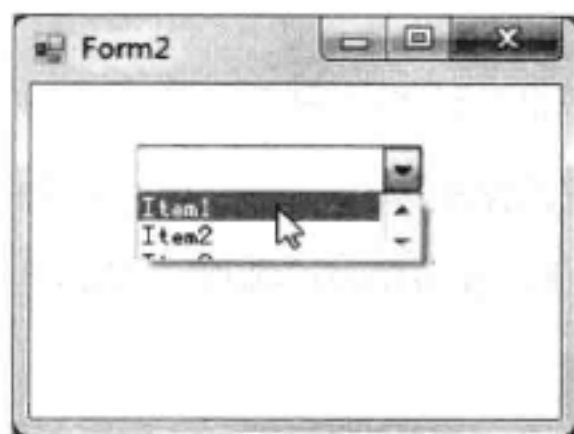


图 14.13 ComboBox 控件

对 DropDownHeight 属性的设置代码如下所示。

```
private void SetDroppedDownHeight()
{
    //设置 ComboBox 控件的下拉框高度
    this.comboBox1.DropDownHeight = 200;
}
```

DropDownWidth 属性： 是一个 int 类型的属性值，用来指示 ComboBox 控件中下拉列表的宽度。在默认情况下，下拉列表的宽度与 ComboBox 控件的宽度相等，当下拉列表中的内容宽度大于下拉列表的宽度时，下拉列表里各项的内容不能完全显示出来，如图 14.14 所示。可以通过设置 DropDownWidth 属性来改变这一情况，具体代码如下所示。

```
private void SetDropDownWidth()
{
    //设置 ComboBox 控件的下拉框宽度
    this.comboBox1.DropDownWidth = 200;
}
```

运行结果如图 14.15 所示。

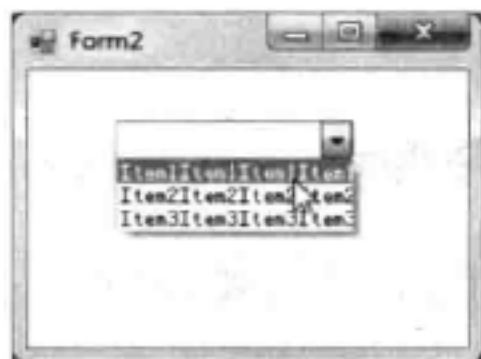


图 14.14 ComboBox 控件

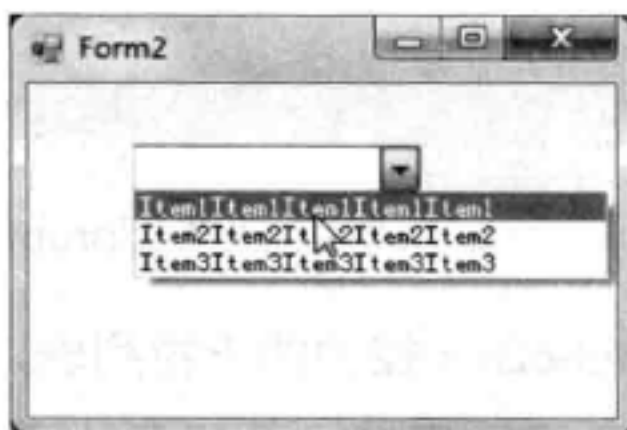


图 14.15 ComboBox 控件

MaxDropDownItems 属性: 是一个 int 类型的属性值, 用来指示 ComboBox 控件中下拉列表的可显示项的最大数量。当 ComboBox 控件中 Items 属性里的项的数量大于 MaxDropDownItems 属性所指示的值时, ComboBox 控件的下拉列表将显示一对方向箭头, 方便用户浏览可选项。可以在属性窗体中, 对 MaxDropDownItems 属性的值进行设置, 也可以使用如下代码进行设置:

```
private void SetMaxDropDownItems()
{
    //设置 ComboBox 控件的下拉框显示的项数
    this.comboBox1.MaxDropDownItems= 5;
}
```

说明: DropDownHeight 属性和 MaxDropDownItems 属性发生冲突时, 以 MaxDropDownItems 属性为标准。

14.2.3 设置 ComboBox 的自动补齐

在用户使用 ComboBox 控件时, 如果允许用户在控件的文本框中输入字符串, 那么在大多数的应用程序中, 都提供对输入的字符串进行自动匹配并补齐的功能。也就是说, 当用户在文本框部分输入字符串时, 程序会根据用户输入的部分与下拉列表中的数据项进行前段匹配, 并使查找到的第一个匹配项自动出现在文本框中, 完成字符串的自动输入。如果所自动匹配出的字符串与用户想要输入的字符串不符的话, 用户可以继续输入, 直到匹配出用户希望选择的选项。若用户输入的字符串与下拉框中各项都不匹配的话, 则程序不会自动补齐字符串。

如果希望 ComboBox 控件使用时具有上述功能, 需要对 ComboBox 控件的 AutoCompleteCustomSource 属性、AutoCompleteSource 属性和 AutoCompleteMode 属性进行设置, 下面将对这 3 个属性分别进行说明。

14.2.4 ComboBox 自动补齐的数据源

AutoCompleteCustomSource 属性用于定义在控件执行自动完成功能时, 所需要进行匹配的数据源。对于 AutoCompleteCustomSource 属性的定义可以在 ComboBox 控件的属性窗体中进行定义。单击 AutoCompleteCustomSource 属性右侧的省略号图标, 打开相应的“字符串集合编辑器”, 如图 14.16 所示。在字符串集合编辑器中输入相应的字符串作为控件

在完成自动填充功能时的查找数据源。




图 14.16 字符串集合编辑器

也可以使用代码对 `AutoCompleteCustomSource` 属性进行设置，具体代码如下所示。

```
private void setAutoCompleteCustomSource()
{
    //向 ComboBox 控件的下拉框添加子项
    string[] customRange = new string[] { "CustomItem1", "CustomItem2",
                                           "CustomItem3", "CustomItem4",
                                           "CustomItem5", "CustomItem6" };
    this.comboBox1.AutoCompleteCustomSource.AddRange(customRange);
}
```

`AutoCompleteSource` 属性用于指定与用户输入数据进行匹配的数据集合。`AutoCompleteSource` 值是一个 `AutoCompleteSource` 枚举类型值，在上文所描述的功能实现中，需要将 `AutoCompleteSource` 属性值设置为 `AutoCompleteSource.ListItems`。

 **注意：** `AutoCompleteCustomSource` 属性和 `AutoCompleteSource` 属性同样适用于 `TextBox` 控件中。

在 `ComboBox` 控件中，这个枚举值也是最常使用的一个枚举值。对于具有此属性的其他控件，如 `TextBox` 控件，还可以设置 `AutoCompleteSource` 属性值为 `AutoCompleteSource` 枚举类型的其他枚举值。`AutoCompleteSource` 枚举类型的其他常用值还有下面几种。

- ❑ `AutoCompleteSource.FileSystem`: 当用户在控件中输入字符串时，控件将输入的字符串与计算机的文件系统自动进行匹配，并根据自动完成模式所指定的自动完成方法对用户所输入的字符串进行自动补充。当 `AutoCompleteSource` 属性值为此枚举值时，程序的运行效果如图 14.17 所示。

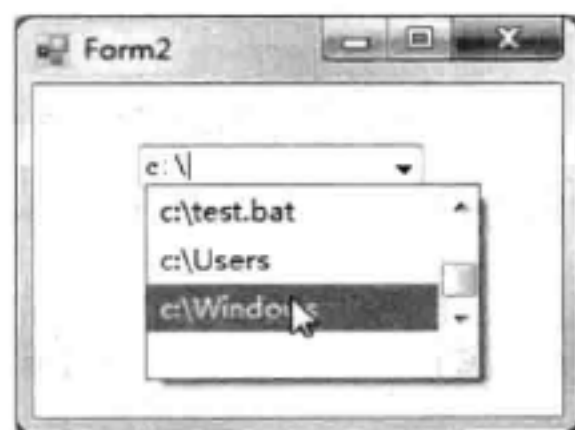


图 14.17 文件系统作为自动填充数据源

- ❑ `AutoCompleteSource.FileSystemDirectories`: `AutoCompleteSource` 枚举类型的这个枚

举值与 `AutoCompleteSource.FileSystem` 枚举值的功能类似。区别在于当 `AutoCompleteSource` 属性值为 `AutoCompleteSource.FileSystem` 时,字符串是与计算机文件系统中的所有文件和文件夹进行匹配,而当 `AutoCompleteSource` 属性值为 `AutoCompleteSource.FileSystemDirectories` 时,字符串仅是计算机文件系统中的所有文件夹进行匹配。

- ❑ `AutoCompleteSource.RecentlyUsedList`: 当用户在控件中输入字符串时,控件将输入的字符串与用户近期访问的 URL 进行匹配,并根据自动完成模式所指定的自动完成方法对用户所输入的字符串进行自动补充。
- ❑ `AutoCompleteSource.HistoryList`: 当用户在控件中输入字符串时,控件将输入的字符串与用户访问的 URL 历史数据进行匹配,并根据自动完成模式所指定的自动完成方法对用户所输入的字符串进行自动补充。
- ❑ `AutoCompleteSource.AllUrl`: 当用户在控件中输入字符串时,控件将用户近期访问的 URL 和用户访问的 URL 历史数据作为数据源,与用户输入的字符串进行匹配,并根据自动完成模式所指定的自动完成方法对用户所输入的字符串进行自动补充。当 `AutoCompleteSource` 属性值为此枚举值时,程序的运行效果如图 14.18 所示。
- ❑ `AutoCompleteSource.AllSystemSources`: 当用户在控件中输入字符串时,控件将 `AutoCompleteSource.AllUrl` 和 `AutoCompleteSource.FileSystem` 所指示的数据源的合集作为数据源,与用户输入的字符串进行匹配,并根据自动完成模式所指定的自动完成方法对用户所输入的字符串进行自动补充。
- ❑ `AutoCompleteSource.CustomSource`: 当用户在控件中输入字符串时,控件将输入的字符串与用户自定义的数据源进行匹配,即与控件的 `AutoCompleteCustomSource` 属性所指示的字符串数组进行匹配,并根据自动完成模式所指定的自动完成方法对用户所输入的字符串进行自动补充。当 `AutoCompleteSource` 属性值为此枚举值时,程序的运行效果如图 14.19 所示。

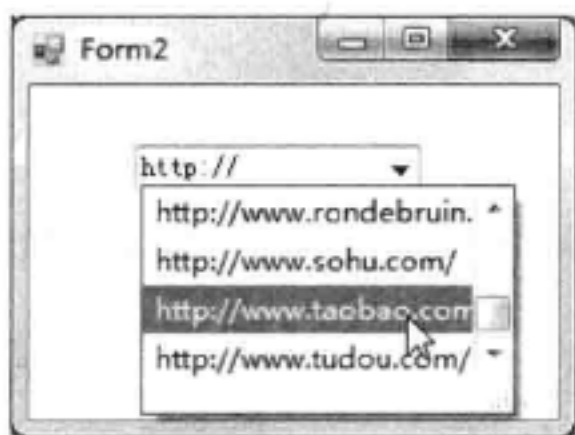


图 14.18 URL 作为自动填充数据源

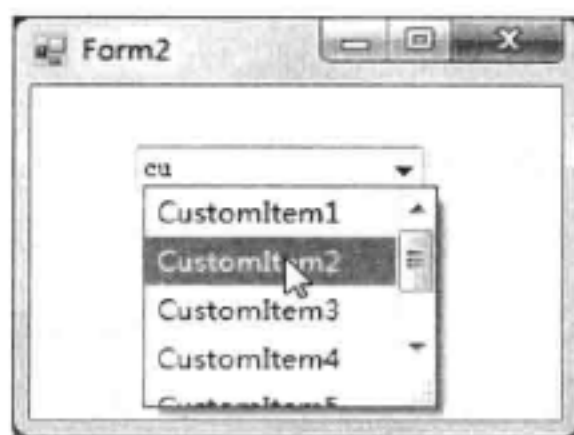


图 14.19 用户自定义数据源作为自动填充数据源

技巧: 通过对 `AutoCompleteSource` 属性进行设置,可以使应用程序更具有用户友好性。

开发人员可以在属性窗体里对 `AutoCompleteSource` 属性进行设置,也可以使用编程方式对此属性进行设置,具体代码如下所示。

```
private void setAutoCompleteSource()
{
    //设置 ComboBox 控件的自动补齐数据源
```



```
this.comboBox1.AutoCompleteSource = AutoCompleteSource.ListItems;
}
```

14.2.5 设置 ComboBox 自动补齐方式

在对控件的 `AutoCompleteSource` 属性进行设置后, 还要对控件的 `AutoCompleteMode` 属性进行正确的设置, 控件才能够在用户输入时, 具有自动填充功能。

`AutoCompleteMode` 属性值是一个 `AutoCompleteMode` 枚举类型的值, 根据开发人员所设置的值不同, 控件的自动完成模式也不相同。`AutoCompleteMode` 枚举类型的具体成员值如下所示。

- ❑ `AutoCompleteMode.None`: 当用户在控件中输入字符串时, 控件并不会将输入的字符串与 `AutoCompleteSource` 属性值所指定的数据源中的数据进行匹配。
- ❑ `AutoCompleteMode.Suggest`: 当用户在控件中输入字符串时, 控件将输入的字符串与 `AutoCompleteSource` 属性值所指定的数据源中的数据进行匹配, 并将匹配出的内容自动添加到控件下方的列表中, 具体效果如图 14.20 所示。
- ❑ `AutoCompleteMode.Append`: 当用户在控件中输入字符串时, 控件将输入的字符串与 `AutoCompleteSource` 属性值所指定的数据源中的数据进行匹配, 并将匹配出的项按照字典序排列后, 将第一项自动添加到用户已输入的字符串的后面, 具体效果如图 14.21 所示。
- ❑ `AutoCompleteMode.SuggestAppend`: 当用户在控件中输入字符串时, 控件将输入的字符串与 `AutoCompleteSource` 属性值所指定的数据源中的数据进行匹配, 并将匹配出的内容自动添加到控件下方的列表中, 同时将列表中的第一项添加到用户已输入的字符串的后面, 具体效果如图 14.22 所示。

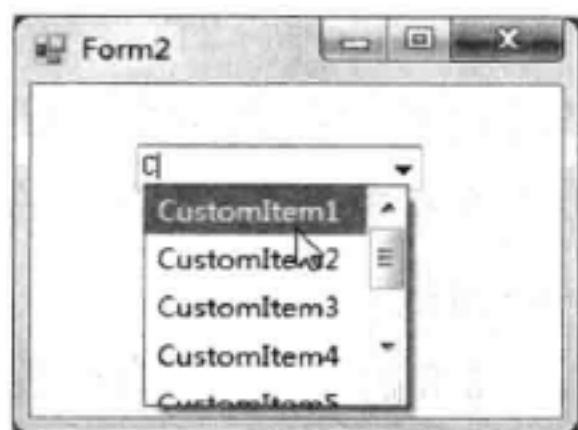


图 14.20 Items 属性作为自动填充数据源

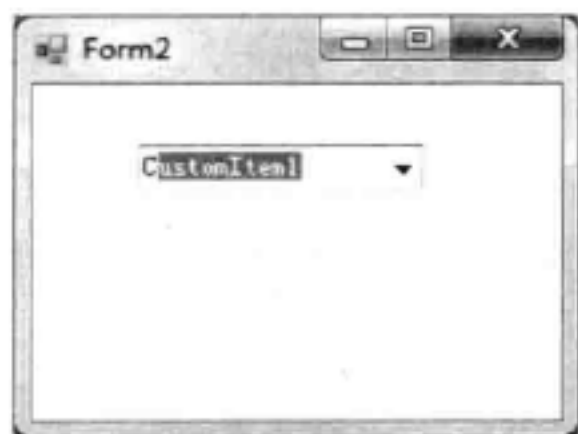


图 14.21 自动完成模式值为 Append

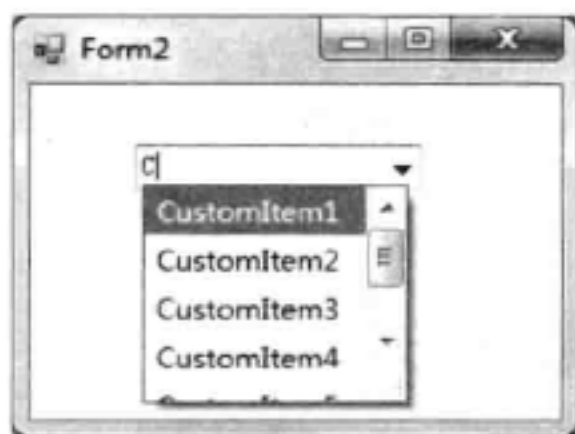


图 14.22 自动完成模式值为 SuggestAppend

技巧: 在应用程序开发中, 如果用户没有特别说明, 那么推荐使用如图 14.22 所示的补齐方式。

程序员可以在属性窗体里对 `AutoCompleteMode` 属性进行设置, 也可以使用编程方式对此属性进行设置, 具体代码如下所示。

```
private void setAutoCompleteMode()  
{  
    //设置 ComboBox 控件的自动补齐模式  
    this.comboBox1.AutoCompleteMode = AutoCompleteMode.SuggestAppend;  
}
```


14.2.6 ComboBox 的常见事件

当用户对 ComboBox 控件进行使用时,所做的操作会引发控件不同的事件,开发人员则可以根据需求,在这些事件的处理函数中添加处理机制,完成控件的逻辑功能。下面将对 ComboBox 控件的一些常用事件进行简单的介绍。

当用户单击 ComboBox 控件右侧的下拉按钮时,将展开 ComboBox 控件的下拉框部分,在显示此下拉框部分时,将触发 DropDown 事件。同样的,当用户在下拉框部分进行选择后,空间的下拉框部分将关闭,此时,会触发控件的 DropDownClosed 事件。当应用程序中,希望下拉框展开或消失的时候,进行某种逻辑操作时,则可以在这个事件的事件处理函数中添加程序代码。

在下面的代码中,将在控件的 DropDown 事件和 DropDownClosed 事件中添加相应的处理程序,使得下拉框在展开或消失的时候弹出相应的消息对话框。

```
//当展开 ComboBox 控件的下拉框时被触发  
private void comboBox1_DropDown(object sender, EventArgs e)  
{  
    MessageBox.Show("DropDown 事件被触发");  
}  
//当关闭 ComboBox 控件的下拉框时被触发  
private void comboBox1_DropDownClosed(object sender, EventArgs e)  
{  
    MessageBox.Show("DropDownClosed 事件被触发");  
}
```

 **技巧:** DropDown 事件和 DropDownClosed 事件属于频繁触发事件,尽量不要在其处理函数中添加需要大量运行时间的处理逻辑。

在展开下拉框时,将弹出如图 14.23 所示的消息对话框。在关闭下拉框时,将弹出如图 14.24 所示的消息对话框。



图 14.23 DropDown 事件被触发



图 14.24 DropDownClosed 事件被触发

14.2.7 修改 ComboBox 的子项

在用户对 ComboBox 控件中的所选项进行更改时,将引发控件的 `SelectedIndexChanged` 事件。

在下面的应用程序中,对 `SelectedIndexChanged` 事件的事件处理程序进行编辑,使应用程序可以在用户选定某项后,自动将下拉菜单中所有与所选项内容相同的项都删除,同时在相应的文本框中显示删除项的内容和删除的个数,代码如下:

```
//在这个方法中将初始化一个 ComboBox 控件
//并将此控件添加到窗体中去
private void InitializeComboBox()
{
    ComboBox comboBox1 = new System.Windows.Forms.ComboBox();
    //使用嵌套循环向 ComboBox 控件添加可选项
    //嵌套循环完成后,comboBox1 中的下拉列表的项中的内容为
    //item0 99 个
    //item1 98 个
    //item2 97 个
    //item3 96 个
    //...
    for (int i = 0; i < 100; i++)
    {
        for (int j = 0; j < i; j++)
        {
            comboBox1.Items.Add("item " + j.ToString());
        }
    }
    //设置 comboBox1 的其他属性
    comboBox1.Location = new System.Drawing.Point(14, 50);
    comboBox1.MaxDropDownItems = 5;
    comboBox1.DropDownStyle = ComboBoxStyle.DropDownList;
    comboBox1.Size = new System.Drawing.Size(100, 75);
    //将 comboBox1 添加到窗体控件中去
    this.Controls.Add(comboBox1);
    //注册 comboBox1_SelectedIndexChanged 事件
    comboBox1.SelectedIndexChanged += new
        System.EventHandler(comboBox1_SelectedIndexChanged);
}
//当 comboBox1 控件中的所选项发生变化时触发该事件
private void comboBox1_SelectedIndexChanged(object sender, System.
EventArgs e)
{
    ComboBox comboBox = (ComboBox) sender;
    //获得 ComboBox 控件中当前的所选项
    string selectedItem = (string)comboBox.SelectedItem;
    //定义 int 型变量 count,
```



```

//用于记录所选项在 ComboBox 控件中与所选项内容相同的项的个数
int count = 0;
//定义 int 型变量 resultIndex
//用于记录所选项在 ComboBox 控件中与所选项内容相同的第一个项的索引
int resultIndex = -1;
//使用 FindStringExact() 方法获取在 ComboBox 控件中与所选项内容相同的第一个项的索引
resultIndex = comboBox.FindStringExact(selectedItem);
//判断是否有与 ComboBox 控件中所选项内容相同的项
while (resultIndex != -1)
{
    //删除与所选项内容相同的项, 计数器加 1
    comboBox.Items.RemoveAt(resultIndex);
    count += 1;
    resultIndex = comboBox.FindStringExact(selectedItem, resultIndex);
}
textBox1.Text = selectedItem + " 共有" + count + "个";
}

```

程序的运行结果如图 14.25 所示。从图中可以看出, 程序运行后, 用户选择了列表框下拉列表里的“Item7”项, 则在文本框中显示出“Item7 共有 92 个”字样。再看下拉列表中, 所有的文本内容为“Item7”的项已经被全部删除。


 **技巧:** 当对 ComboBox 控件的子项内容进行重新设置时, 可调用 BeginUpdate()方法和 EndUpdate()方法使程序界面不要频繁更新。




图 14.25 程序运行结果

14.2.8 ComboBox 的子项绘制

在程序生成 ComboBox 控件时, 需要对控件进行重新绘制, 这时, 就会触发两个在自定义 ComboBox 控件外观时经常使用到的事件: DrawItem 事件和 MeasureItem 事件。

DrawItem 事件主要是在重新绘制 ComboBox 控件时被触发, 而 MeasureItem 事件则是在描述 ComboBox 控件中的可选项时被触发。在应用程序开发过程中, 可以在这两个事件的处理函数中添加适当的代码, 使 ComboBox 控件中的各项外观具有了用户定制性。

 **技巧:** 当对 ComboBox 控件的子项进行重新绘制时, 可调用 BeginUpdate()方法和 EndUpdate()方法使程序界面不要频繁更新。

在下面的示例代码中, 将对一个 ComboBox 控件中的子项进行重新绘制, 代码设置如下:

```

//声明一个字符串数组, 用于初始化 ComboBox 控件的数据源
private string[] colors;
//初始化 ComboBox 控件
private void InitializeComboBox()

```



```

{
    //实例化一个 ComboBox 控件
    ComboBox comboBox1 = new ComboBox();
    //初始化 ComboBox 控件
    comboBox1.DrawMode = System.Windows.Forms.DrawMode.OwnerDrawVariable;
    comboBox1.Location = new System.Drawing.Point(10, 20);
    comboBox1.Size = new System.Drawing.Size(140, 140);
    comboBox1.DropDownHeight = 300;
    comboBox1.TabIndex = 0;
    comboBox1.DropDownStyle = ComboBoxStyle.DropDown;
    colors = new string[] { "红", "黄", "蓝" };
    comboBox1.DataSource = colors;
    this.Controls.Add(comboBox1);
    //注册事件处理程序
    comboBox1.DrawItem += new DrawItemEventHandler(comboBox1_DrawItem);
    comboBox1.MeasureItem += new MeasureItemEventHandler(comboBox1_
        MeasureItem);
}
//当知道绘制 ComboBox 控件中的各子项大小时被触发
private void comboBox1_MeasureItem(object sender, MeasureItemEventArgs e)
{
    //设置各项的高度
    switch (e.Index)
    {
        case 0:
            e.ItemHeight = 40;
            break;
        case 1:
            e.ItemHeight = 30;
            break;
        case 2:
            e.ItemHeight = 20;
            break;
    }
    //设置各项的宽度
    e.ItemWidth = 100;
}
//当绘制 ComboBox 控件中的各子项时事件被触发
private void comboBox1_DrawItem(object sender, DrawItemEventArgs e)
{
    //初始化变量
    float size = 0;
    System.Drawing.Font font;
    FontFamily family = null;
    System.Drawing.Color color = new System.Drawing.Color();
    //根据子项索引设置各子项格式

```

```

switch (e.Index)
{
    case 0:
        size = 35;
        color = System.Drawing.Color.Red;
        family = FontFamily.GenericMonospace;
        break;
    case 1:
        size = 25;
        color = System.Drawing.Color.Yellow;
        family = FontFamily.GenericMonospace;
        break;
    case 2:
        size = 15;
        color = System.Drawing.Color.Blue;
        family = FontFamily.GenericMonospace;
        break;
}
//绘制下拉框背景
e.DrawBackground();
Rectangle rectangle = new Rectangle(0, e.Bounds.Top, e.Bounds.Height,
e.Bounds.Height);
e.Graphics.FillRectangle(new SolidBrush(color), rectangle);
//绘制下拉框各子项字体
font = new Font(family, size, FontStyle.Regular);
e.Graphics.DrawString(colors[e.Index], font, System.Drawing.Brushes.
Black,
                        new RectangleF(e.Bounds.X + rectangle.Width,
e.Bounds.Y, e.Bounds.Width, e.Bounds.Height));
//绘制下拉框各子项
e.DrawFocusRectangle();
}

```

程序的运行结果如图 14.26 所示。从图中可以看出，程序对 ComboBox 下拉框中的各可选项进行了重新绘制，使得各项前方均添加了颜色不同的方块，在方块后面是对方块颜色的文字说明。



图 14.26 程序运行结果

14.3 CheckedListBox 控件

CheckedListBox（可选列表框）控件类似于 ListBox 和 CheckBox 控件的综合体，允许用户在 ListBox 内有选择地挑选具体内容。

14.3.1 带复选标记的列表控件 CheckedListBox

CheckedListBox 控件在 .NET 中的命名空间为 System.Windows.Forms。在 Windows 应用程序编程中，CheckedListBox 控件几乎具有 ListBox 控件的所有功能，同时 CheckedListBox 控件还可以在列表项的旁边显示复选标记。

CheckedListBox 控件在 Visual Studio 的工具箱中的外观如图 14.27 所示，将 CheckedListBox 控件拖曳到窗体并选定后，如图 14.28 所示。下一步，可在 Visual Studio 的属性窗体中对 CheckedListBox 的属性进行设置，使其达到应用程序的具体需求。

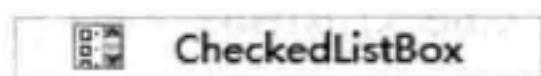


图 14.27 CheckedListBox 控件

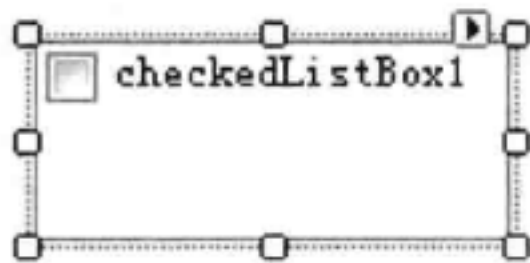


图 14.28 CheckedListBox 控件

注意：CheckedListBox 控件几乎具有 ListBox 控件的所有功能。

对 CheckListBox 控件里的列表进行编辑，在“属性”窗口中，选择 Items 选项，单击相应属性值右侧的...按钮，如图 14.29 所示。



图 14.29 Items 属性

打开相应的“字符串集合编辑器”为 CheckListBox 控件添加项，如图 14.30 所示。在这里，添加了 ItemA、ItemB 和 ItemC 等 3 项，单击“确定”按钮，添加后的效果如图 14.31 所示。



图 14.30 字符串集合编辑器

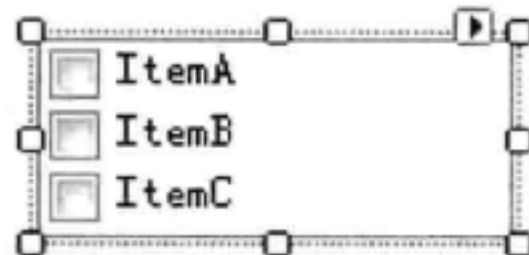


图 14.31 CheckedListBox 控件

既然 CheckedListBox 控件可以看做是每项都是一个 CheckBox 控件的 ListBox, 那么在使用时, CheckedListBox 控件的很多属性、方法和事件也与这两个控件相同。在本节中, 对这些与 CheckBox 和 ListBox 控件相同的属性、事件和方法将不再详细介绍, 这里主要介绍 CheckedListBox 控件的特殊的属性和方法。


在使用 CheckListBox 控件的应用程序编程中, CheckListBox 控件主要用于提供一组并列的可选项, 并将用户的选择结果记录到数据库或根据用户的选择结果决定程序的运行流程。无论作为哪种用途, 应用程序都需要确定并获得 CheckListBox 控件里的各项中的选定项。

可以通过 CheckListBox 控件的 CheckedItems 属性或者 CheckedIndices 属性获得相应的用户选定项。其中 CheckedItems 属性是 CheckListBox 控件中的所有用户选定项的集合, 而 CheckedIndices 属性是 CheckListBox 控件中的所有用户选定项的索引的集合。

在下面的代码中, 即通过 CheckedItems 属性获得了 CheckListBox 控件中的所有选定项, 并且以 MessageBox 的形式展示出来。

```
//判断在 CheckedListBox 控件中是否有选中项
if (checkedListBox1.CheckedItems.Count != 0)
{
    //如果有, 则循环所有的选中项, 并将结果显示出来
    string checkedItem = "";
    for (int x = 0; x <= checkedListBox1.CheckedItems.Count - 1; x++)
    {
        checkedItem = checkedItem + "选中项" + (x+1).ToString() + ": "
            + checkedListBox1.CheckedItems[x].ToString() + "\r\n";
    }
    //以 MessageBox 的形式将结果显示出来
    MessageBox.Show (checkedItem);
}
```

CheckListBox 控件的 CheckOnClick 属性用于标志当用户使用 CheckListBox 控件时, 如何更改 CheckListBox 控件中各项的 Checked 状态。CheckOnClick 属性是一个 boolean 类型的值, 当其值为 true 时, 用户只要选择想要选中或取消选中的项, 就会改变此项的选中状态。当其值为 false 时, 用户若要改变此项的选择状态, 则需要选中 CheckListBox 控件中的每一项前面的空格。

 注意: CheckOnClick 属性相当于 CheckBox 控件的 AutoChecked 属性。

对于 CheckOnClick 的设置,可以在属性窗体里进行设置,也可以使用如下代码进行设置:

```
private void setCheckOnClick()
{
    this.checkedListBox1.CheckOnClick = true;
}
```

CheckListBox 控件也可以像 ListBox 控件一样通过设置 SelectionMode 属性来设置控件的选择模式,但不同的是,在 CheckListBox 控件里,控件并不能设置多选的选择模式。也就是说,对于 CheckListBox 控件,SelectionMode 属性只能设置为 SelectionMode.None 或 SelectionMode.One 两个 SelectionMode 枚举类型值。

如果将 SelectionMode 设置为 SelectionMode.None,则对于 CheckListBox 控件中的各项将不能进行选择,同样用户也不能更改 CheckListBox 控件中各项的选中状态。

对于 SelectionMode 属性的设置,可以在属性窗体里进行设置,也可以使用如下代码进行设置:

```
private void setSelectionMode()
{
    this.checkedListBox1.SelectionMode = SelectionMode.One;
}
```

14.3.2 CheckedListBox 控件编程示例

在下面的这个示例中,将利用 CheckListBox 控件完成一个学生选课的课程设置应用程序。这个应用程序中的执行效果图如图 14.32 所示。

在这个应用程序中,用户可以通过在窗体左侧的 CheckListBox 控件中选择想要的课程。当用户选中的课程数大于等于 1 时,将激活窗体中的“显示所选课程”按钮,用户单击“显示所选课程”按钮,则会将 CheckListBox 控件中选中的项添加到右侧的 ListBox 控件中。此时将激活“保存所选课程”按钮,单击该按钮将保存用户所选的课程,并将 ListBox 控件清空、取消 CheckListBox 控件中所有的选中项。

如果在 CheckListBox 控件中所列举的课程项没有用户想要选择的项时,用户可以通过向窗体中的文本框中输入相应的课程,来向 CheckListBox 控件添加项。当文本框内容为空时,“添加新课程”按钮是不可用的。在用户向文本框中输入信息后,将激活“添加新课程”按钮。此按钮被激活后,单击此按钮,则会将文本框中的文本作为新项添加到 CheckListBox 控件中。

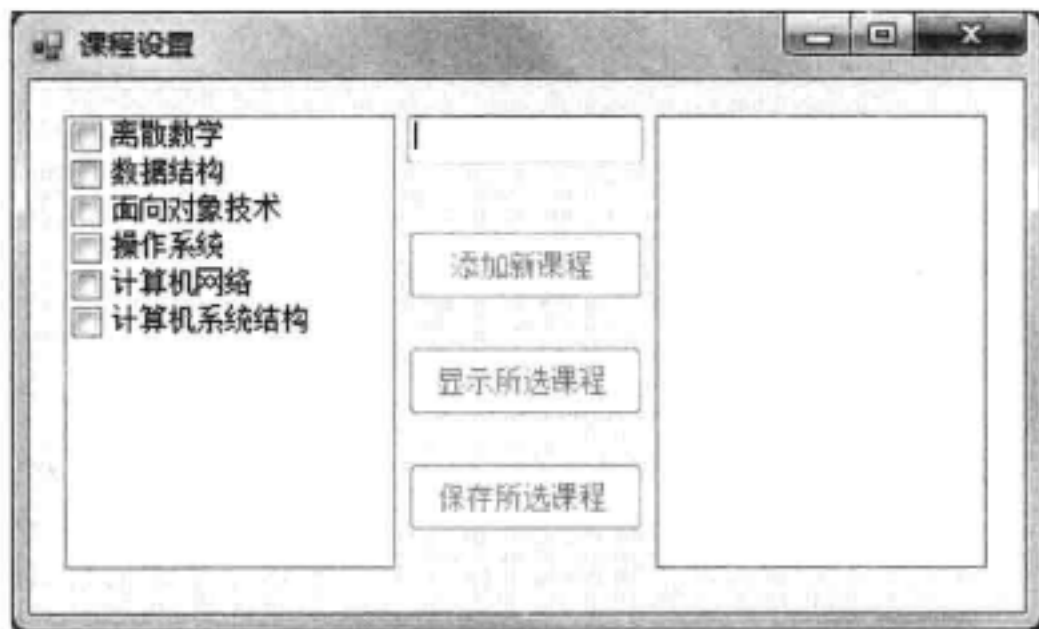


图 14.32 学生选课应用程序运行效果图

说明：这一类型的应用程序，在应用程序开发过程中经常遇见，希望读者认真领会。

程序的具体实现代码如下所示。

```
using System;
//绘制子控件的命名空间
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Linq;
using System.IO;
partial class Form1 : System.Windows.Forms.Form
{
    //设置窗体中的控件
    private System.Windows.Forms.TextBox textBox1;
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.Button button3;
    private System.Windows.Forms.CheckedListBox checkedListBox1;
    private System.Windows.Forms.ListBox listBox1;
    private System.ComponentModel.Container components;
    //Form1 构造函数
    public Form1()
    {
        //初始化控件
        InitializeComponent();
        //设置 CheckedListBox 控件中的项
        string[] myComponent = { "离散数学", "数据结构", "面向对象技术",
                                   "操作系统", "计算机网络", "计算机系统结构" };
        checkedListBox1.Items.AddRange(myComponent);
        //设置 CheckedListBox 控件的 CheckOnClick 属性
        checkedListBox1.CheckOnClick = true;
    }
    //初始化各个控件
    private void InitializeComponent()
    {
        //初始化窗体里控件
        this.components = new System.ComponentModel.Container();
        this.textBox1 = new System.Windows.Forms.TextBox();
        this.button1 = new System.Windows.Forms.Button();
        this.button2 = new System.Windows.Forms.Button();
        this.button3 = new System.Windows.Forms.Button();
        this.checkedListBox1 = new System.Windows.Forms.CheckedListBox();
        this.listBox1 = new System.Windows.Forms.ListBox();
        //设置 TextBox 控件属性
        this.textBox1.Location = new System.Drawing.Point(160, 15);
        this.textBox1.Size = new System.Drawing.Size(100, 20);
        this.textBox1.TabIndex = 0;
        this.textBox1.TextChanged += new System.EventHandler(this.textBox1_
            TextChanged);
        //设置 button1 控件属性
        this.button1.Enabled = false;
        this.button1.Location = new System.Drawing.Point(160, 65);
        this.button1.Size = new System.Drawing.Size(100, 30);
```



```

this.button1.TabIndex = 1;
this.button1.Text = "添加新课程";
this.button1.Click += new System.EventHandler(this.button1_Click);
//设置 button2 控件属性
this.button2.Enabled = false;
this.button2.Location = new System.Drawing.Point(160, 115);
this.button2.Size = new System.Drawing.Size(100, 30);
this.button2.TabIndex = 2;
this.button2.Text = "显示所选课程";
this.button2.Click += new System.EventHandler(this.button2_Click);
//设置 button3 控件属性
this.button3.Enabled = false;
this.button3.Location = new System.Drawing.Point(160, 165);
this.button3.Size = new System.Drawing.Size(100, 30);
this.button3.TabIndex = 3;
this.button3.Text = "保存所选课程";
this.button3.Click += new System.EventHandler(this.button3_Click);
//设置 checkedListBox1 控件属性
this.checkedListBox1.Location = new System.Drawing.Point(15, 15);
this.checkedListBox1.Size = new System.Drawing.Size(140, 200);
this.checkedListBox1.TabIndex = 4;
this.checkedListBox1.ItemCheck += new
    ItemCheckEventHandler(this.checkedListBox1_
        ItemCheck);
//设置 listBox1 控件属性
this.listBox1.Location = new System.Drawing.Point(265, 15);
this.listBox1.Size = new System.Drawing.Size(140, 200);
this.listBox1.TabIndex = 5;
//设置窗体大小
this.ClientSize = new System.Drawing.Size(400, 230);
//将控件添加到属性窗体中
this.Controls.AddRange(new System.Windows.Forms.Control[]
    {this.textBox1,
      this.button1,
      this.button2,
      this.button3,
      this.checkedListBox1,
      this.listBox1});

this.Text = "课程设置";
}
//button1 控件被单击时引发的事件
private void button1_Click(object sender, System.EventArgs e)
{
    //如果 textBox1 控件的文本内容不为空
    if (textBox1.Text != "")
    {
        //如果 checkedListBox1 中的选中项中不包含有 textBox1 中指示的项
        if (checkedListBox1.CheckedItems.Contains(textBox1.Text) ==
            false)
        {
            //则将此项添加到 checkedListBox1 中
            checkedListBox1.Items.Add(textBox1.Text, CheckState.Checked);
        }
    }
}

```



```
    }
    textBox1.Text = "";
}
}
//textBox1 控件中文本更改时引发的事件
private void textBox1_TextChanged(object sender, System.EventArgs e)
{
    //如果 textBox1 控件中的文本为空, 则 button1 控件不可用
    if (textBox1.Text == "")
    {
        button1.Enabled = false;
    }
    //如果 textBox1 控件中的文本不为空, 则 button1 控件可用
    else
    {
        button1.Enabled = true;
    }
}
//button2 控件被单击时引发的事件
private void button2_Click(object sender, System.EventArgs e)
{
    //清空 listBox1 控件中的各项
    listBox1.Items.Clear();
    button3.Enabled = false;
    //将 checkedListBox1 中的选中项添加到 listBox1 中
    for (int i = 0; i < checkedListBox1.CheckedItems.Count; i++)
    {
        listBox1.Items.Add(checkedListBox1.CheckedItems[i]);
    }
    //当 listBox1 中的子项数大于 0 时, button3 控件可用
    if (listBox1.Items.Count > 0)
    {
        button3.Enabled = true;
    }
}
//当 checkedListBox1 控件中的各项的选中状态发生变化时发生
private void checkedListBox1_ItemCheck(object sender, ItemCheckEventArgs e)
{
    //当 CheckListBox 中的选中项不为 0 时, 激活“显示所选课程”按钮
    if (e.NewValue == CheckState.Unchecked)
    {
        if (checkedListBox1.CheckedItems.Count == 1)
        {
            button2.Enabled = false;
        }
    }
    else
    {
        button2.Enabled = true;
    }
}
```



```

private void button3_Click(object sender, System.EventArgs e)
{
    //将选中项保存, 清空 ListBox 控件和 TextBox 控件, 取消 CheckListBox 中的所
    //有的选中项
    listBox1.Items.Clear();
    //IEnumerator 是所有非泛型枚举数的基接口
    IEnumerator enumerator;
    enumerator = checkedListBox1.CheckedIndices.GetEnumerator();
    int index;
    //当 checkedListBox1 中的选中项未循环遍历时
    while (enumerator.MoveNext() != false)
    {
        index = (int)enumerator.Current;
        checkedListBox1.SetItemChecked(index, false);
    }
    button3.Enabled = false;
}
}

```

14.4 ListView 控件

ListView 控件可使用 4 种不同视图显示项目。通过此控件, 可将项目组成带有或不带有列标头的列, 并显示伴随的图标和文本。可使用 ListView 控件将称做 ListItem 对象的列表条目组织成 4 种不同的视图之一。

14.4.1 带图标的列表控件 ListView

ListView 控件在 .NET 中命名空间为 System.Windows.Forms。在 Windows 应用程序编程中, ListView 控件可以为用户显示带图标的项的列表, 就像在 Windows 操作系统中的 Windows 资源管理器功能的左窗格中, 显示每个文件夹中的内容一样, 如图 14.33 所示。



图 14.33 Windows 资源管理器

ListView 控件在 Visual Studio 的工具箱中的图示如图 14.34 所示, 将 ListView 控件拖曳到窗体选定后, 如图 14.35 所示。下一步, 可在 Visual Studio 的属性窗体中对 ListView 控件的属性进行设置, 使其达到应用程序的具体需求。



图 14.34 ListView 控件

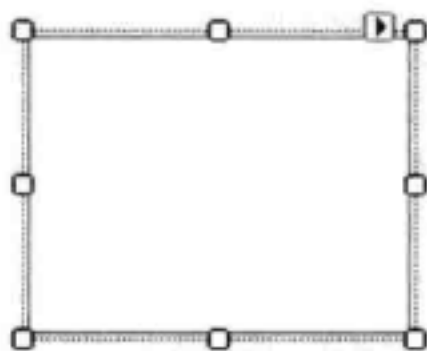



图 14.35 ListView 控件

14.4.2 ListView 的 5 种视图

ListView 控件可以显示可用的 5 种不同视图之一。这 5 种视图模式分别为 LargeIcon、SmallIcon、List、Tile 和 Details, 而这 5 种视图模式的确定是由 ListView 控件的 View 属性控制的。

 **注意:** 根据操作系统的版本不同, 有些视图不能正确显示。

- ❑ 当 View 属性的值为 LargeIcon 时, ListView 控件显示为“大图标视图模式”, 如图 14.36 所示, ListView 控件中的每一项都以一个大图标和一个说明项的文本组成。
- ❑ 当 View 属性的值为 SmallIcon 时, ListView 控件显示为“小图标视图模式”, 如图 14.37 所示, ListView 控件中的每一项都以一个小图标和一个说明项的文本组成。
- ❑ 当 View 属性的值为 List 时, ListView 控件显示为“列表视图模式”, 如图 14.38 所示, ListView 控件中的每一项都以一个小图标和一个说明项的文本组成, 并且所有的项均在单列中显示。



图 14.36 LargeIcon 视图



图 14.37 SmallIcon 视图



图 14.38 List 视图

- ❑ 当 View 属性的值为 Details 时, ListView 控件显示为“详细信息视图模式”, 如图 14.39 所示, ListView 控件中的每一项都以一个主项和与其相关的一系列子项组成, 并且 ListView 控件中的每一行只能显示一项。
- ❑ 当 View 属性的值为 Tile 时, ListView 控件显示为“平铺视图模式”, 如图 14.40 所示, ListView 控件中的每一项都以一个大图标、一个主项中的文本内容和与其相关的一系列子项组成。



图 14.39 Details 视图




图 14.40 Title 视图

在 ListView 控件的属性中, 有一些属性只对 ListView 控件的某种视图模式有效, 对于一些常用的属性所对应的有效视图如表 14.3 所示。

表 14.3 ListView 控件常用属性所对应视图

ListView 控件常用属性	LargeIcon	SmallIcon	List	Tile	Details
Alignment 属性	✓	✓			
AllowColumnReorder 属性					✓
AutoArrange 属性	✓	✓			
CheckBoxes 属性	✓	✓	✓		✓
Columns 属性				✓	✓
FocresdItem 属性	✓	✓	✓	✓	✓
FullRowSelect 属性					✓
GridLines 属性					✓
Groups 属性	✓	✓		✓	✓
InsertionMark 属性	✓	✓		✓	
Items 属性	✓	✓	✓	✓	✓
LabelEdit 属性	✓	✓	✓	✓	✓
LabelWrap 属性	✓	✓			
LargeImageList 属性	✓			✓	
MultiSelect 属性	✓	✓	✓	✓	✓
SelectedIndices 属性	✓	✓	✓	✓	✓
SelectedItems 属性	✓	✓	✓	✓	✓
ShowGroups 属性	✓	✓		✓	✓
SmallImageList 属性		✓	✓		✓
StateImageList 属性	✓	✓	✓		✓
TileSize 属性				✓	
TopItem 属性			✓		✓

 注意: 在 ListView 控件的属性中, 有一些属性只对 ListView 控件的某种视图模式有效。

14.4.3 使用任务窗体设置 ListView

在窗体设计器中, 单击窗体右上角的一个向右的小箭头, 可以打开 ListView 控件的 ListView 任务窗体, 如图 14.41 所示。读者可以通过对此任务窗体中的任务进行设置, 从

而完成 ListView 控件的初始化工作。

在 ListView 任务窗体中，主要可以完成以下几个任务：

- ☐ 编辑项。
- ☐ 编辑列。
- ☐ 编辑组。
- ☐ 选择视图。
- ☐ 设置相应图片信息。




图 14.41 ListView 任务窗体

通常来说，在 Windows 应用程序中的 ListView 控件里，每一项都有一个相对应的图标，这个图标的大小类型是由 ListView 控件的视图模式来决定的。而每一个项中所显示的图标，则是 ListView 控件的 LargeImageList 属性和 SmallImageList 属性所指定的 ImageList 组件中的图标。

14.4.4 用 ImageList 给 ListView 提供图标

在使用 ListView 控件时，开发人员首先需要向 Windows 窗体中添加至少两个 ImageList 组件。对于 ImageList 组件的使用将在后面的章节进行具体的介绍，这里只介绍此组件简单的添加方法。ImageList 组件是一个存储及管理图片集合的组件，其在 Visual Studio 的工具箱中的图示如图 14.42 所示。

 **注意：**在使用 ListView 控件时，开发人员首先需要向 Windows 窗体中添加至少两个 ImageList 组件。

ImageList 组件不能添加在窗体中，它将出现在窗体设计器的底部，如图 14.43 所示。



图 14.42 ImageList 组件

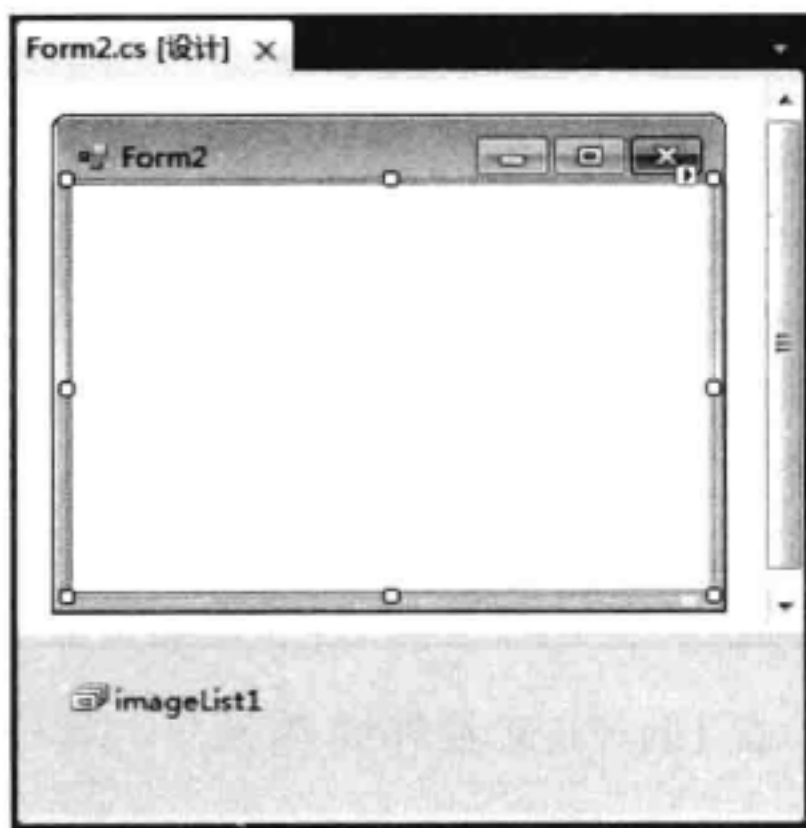


图 14.43 ImageList 组件

对 ImageList 组件的 Images 属性进行编辑，打开相应的图像集合编辑器，如图 14.44 所示，在此对话框中添加在应用程序窗体中用到的图片，以方便在以后的编程中进行引用。在本例中，将对 ImageList 组件控件中添加 8 个图像。

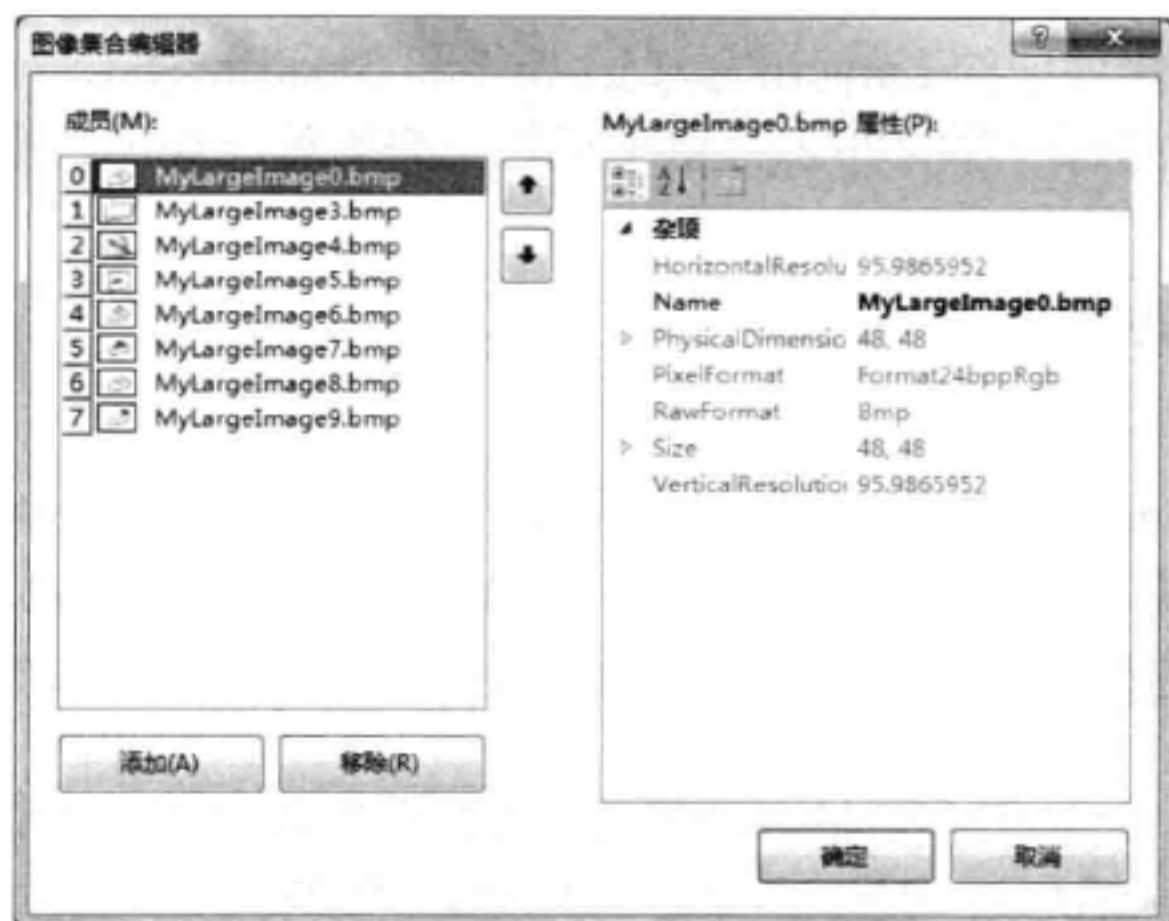


图 14.44 图像集合编辑器

也可以使用代码向 ImageList 组件添加图片，具体代码如下所示。

```
imageList1.ImageStream=(ImageListStreamer)(resources.GetObject("imageList1.ImageStream"));
imageList1.TransparentColor = System.Drawing.Color.Transparent;
//向 imageList1 中添加图片
imageList1.Images.SetKeyName(0, "MyLargeImage0.bmp");
imageList1.Images.SetKeyName(1, "MyLargeImage3.bmp");
imageList1.Images.SetKeyName(2, "MyLargeImage4.bmp");
imageList1.Images.SetKeyName(3, "MyLargeImage5.bmp");
imageList1.Images.SetKeyName(4, "MyLargeImage6.bmp");
imageList1.Images.SetKeyName(5, "MyLargeImage7.bmp");
imageList1.Images.SetKeyName(6, "MyLargeImage8.bmp");
imageList1.Images.SetKeyName(7, "MyLargeImage9.bmp");
```

在设置好了一个 ImageList 组件后，便可以使用此组件给 ListView 控件中的相应属性赋值。在本例中，可将 LargeImageList 属性设置为定义好的 ImageList 组件，同时，读者也可自己定义一个 ImageList 组件，并将这个组件赋值给 ListView 控件的 SmallImageList 属性。

注意：向赋值给 SmallImageList 属性的 ImageList 组件中添加图片时，需要添加像素较小的图片，因为 SmallImageList 属性所指定的 ImageList 组件，是为 ListView 控件的 List 视图、Details 视图和 SmallIcon 视图提供图片的，观察图 14.38、图 14.39 和图 14.40，可以发现在这 3 种视图中所显示的图标都较小。

读者也可以在 ListView 任务窗体中对 ListView 控件的属性进行设置，或者使用如下代码进行设置：

```
private void setImageList()
{
    this.listView1.LargeImageList = this.imageList1;
    this.listView1.SmallImageList = this.imageList2;
}
```

设置完 ListView 控件中的 LargeImageList 属性和 SmallImageList 属性后, 则可以通过设置 ListView 控件中的每一项的 ImageIndex 属性, 来为每一项设置相关联的图标。

14.4.5 设置 ListView 的子项

单击 ListView 任务窗体中的“编辑项”文本, 打开“ListViewItem 集合编辑器”窗体, 如图 14.45 所示, 在此窗体中对 ListView 控件中的子项进行设置。也可以通过在“属性”窗口中, 单击 Items 属性旁的...按钮打开“ListViewItem 集合编辑器”窗体。



图 14.45 ListViewItem 集合编辑器

如果使用代码向 ListView 控件添加项, 那么无论当前的 ListView 控件使用的是哪一种视图, 都将使用 Items 属性的 Add() 方法。具体添加方法如下所示。

```
listView1.Items.Add("文件夹 1");
listView1.Items.Add("文件夹 2");
listView1.Items.Add("文本文件 1");
listView1.Items.Add("文本文件 2");
listView1.Items.Add("图片文件 1");
listView1.Items.Add("图片文件 2");
```

从 ListView 控件移除项, 可使用 Items 属性的是 RemoveAt() 或 Clear() 方法。RemoveAt() 方法用于移除指定索引的项, 而 Clear() 方法移除列表中的所有项。具体使用方法如下所示。


```
// 移除列表中的第一项
listView1.Items.RemoveAt(0);
// 清除所有项
listView1.Items.Clear();
```


14.4.6 设置 Details 视图模式的数据

在图 14.40 所示的 Details 视图模式中, ListView 控件可以为每一项都显示为多个列, 例如, 文件列表可以显示文件名、文件类型、大小和文件的上次修改日期。通过单击“ListView 任务”窗体中的“编辑列”文本标签, 可以打开 ListView 控件的“ColumnHeader 集合编辑器”窗体, 如图 14.46 所示。在此窗体中, 向 ListView 控件的 Details 视图模式中添加列头。



图 14.46 ColumnHeader 集合编辑器

 说明: Columns 属性只对 Details 视图有效。

也可以通过调用 Columns 属性的 Add() 方法, 对列表视图中的列进行添加, 具体使用方法如下所示。

```
// 设置为显示详细信息视图
listView1.View = View.Details;
// 添加一列标题为“文件名”, 宽度为 60, 左靠齐方式的列表
listView1.Columns.Add("文件名", 60, HorizontalAlignment.Left);
listView1.Columns.Add("文件类型", 70, HorizontalAlignment.Left);
listView1.Columns.Add("文件大小", 70, HorizontalAlignment.Left);
listView1.Columns.Add("最后修改时间", 140, HorizontalAlignment.Left);
```

添加后的运行效果如图 14.47 所示。



图 14.47 ListView 控件列表视图

在添加好 Details 视图的列表头以后, 可以通过向 ListView 控件中的各项添加子项来完成列表。在 ListViewItem 集合编辑器里单击 SubItems 属性旁的... 按钮打开“ListViewSubItem 集合编辑器”窗体, 如图 14.48 所示, 单击“添加”按钮添加 3 个子项, 然后向各子项添加相应的数据。



图 14.48 ListViewItem 集合编辑器

也可以通过调用项的 SubItems 属性返回集合 Add() 方法。

注意: SubItems 属性所定义的内容, 在 Tile 视图中, 将平铺在项图标的右侧。

下面的代码实例为一个列表项设置文件名称、文件类型、文件大小和最后修改时间。代码设置如下所示。

```
//给列表添加一项, 并添加相应的三个子项
listView1.Items.Add("Program Files");
listView1.Items[1].SubItems.Add("文件夹");
listView1.Items[1].SubItems.Add("14.9G");
listView1.Items[1].SubItems.Add("2008-01-01 14:00");
//给列表添加一项, 并添加相应的三个子项
listView1.Items.Add("Documents and Settings");
listView1.Items[0].SubItems.Add("文件夹");
listView1.Items[0].SubItems.Add("421MB");
listView1.Items[0].SubItems.Add("2008-01-01 14:00");
```

添加后的运行效果如图 14.49 所示。

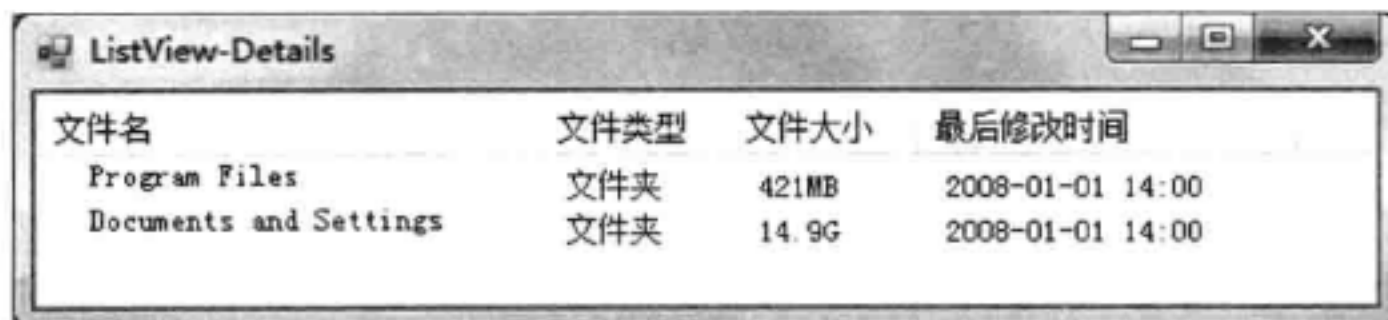


图 14.49 ListView 控件列表视图

14.4.7 给 ListView 添加分组设置

单击 ListView 任务窗体中的“编辑组”文本标签，则可以打开“ListViewGroup 集合编辑器”窗体，如图 14.50 所示，在此窗体中可以对 ListView 控件中的各项添加分组标签。



图 14.50 ListViewGroup 集合编辑器

添加过分组标签后，可以将各项的 Group 属性设置为此项所属组的组索引号。也可以使用代码向 ListView 控件添加分组。具体代码如下所示。

```
//初始化 ListView 控件的子项分组
ListViewGroup listViewGroup1 = new ListViewGroup("文件夹", HorizontalAlign
ment.Left);
ListViewGroup listViewGroup2 = new ListViewGroup("文本文件", HorizontalAlign
ment.Left);
ListViewGroup listViewGroup3 = new ListViewGroup("图片文件", HorizontalAlign
ment.Left);
//初始化 listViewGroup1
listViewGroup1.Header = "文件夹";
listViewGroup1.Name = "listViewGroup1";
//初始化 listViewGroup2
listViewGroup2.Header = "文本文件";
listViewGroup2.Name = "listViewGroup2";
//初始化 listViewGroup3
listViewGroup3.Header = "图片文件";
listViewGroup3.Name = "listViewGroup3";
//将各分组添加至 listView1
this.listView1.Groups.AddRange(new ListViewGroup[] {listViewGroup1,
listViewGroup2,listViewGroup3});
//设置 ListView 控件的子项分组
listViewItem1.Group = listViewGroup1;
listViewItem2.Group = listViewGroup1;
listViewItem3.Group = listViewGroup2;
listViewItem4.Group = listViewGroup2;
```

```
listViewItem5.Group = listViewGroup3;
listViewItem6.Group = listViewGroup3;
```

注意：并不是所有的 ListView 控件的视图模式都适用于使用列表项分组，适合分组的视图有“LargeIcon 视图模式”、“SmallIcon 视图模式”、“Tile 视图模式”和“Details 视图模式”。

实现分组后的 ListView 控件如图 14.51、图 14.52、图 14.53 和图 14.54 所示。



图 14.51 LargeIcon 视图



图 14.52 SmallIcon 视图



图 14.53 Details 视图



图 14.54 Tile 视图

14.4.8 让 ListView 支持拖曳操作

在应用程序中使用 ListView 控件时，除了设置 ListView 控件的视图模式及其中的各列表项外，还需要向 ListView 控件添加一定的功能，其中比较常用的是在 ListView 控件中添加拖曳的功能。在下面的代码示例中用户可对 ListView 控件中的各列进行拖曳，使各项的排列顺序进行改变。具体的代码实现如下所示。

技巧：用户可以在应用程序中，通过拖曳 ListView 控件中的各列表项，而改变各项的排列位置。


```

//初始化 listView1.
private void InitializeListView()
{
    listView1.ListViewItemSorter = new ListViewItemComparer();
    //初始化插入标记
    listView1.InsertionMark.Color = Color.Red;
    //设置 listView1 可拖曳
    listView1.AllowDrop = true;
}
//当一个项目拖曳时启动拖曳操作
private void listView1_ItemDrag(object sender, ItemDragEventArgs e)
{
    listView1.DoDragDrop(e.Item, DragDropEffects.Move);
}
private void listView1_DragEnter(object sender, DragEventArgs e)
{
    //允许在 listView1 控件中进行拖放
    e.Effect = e.AllowedEffect;
}
//像拖曳项目一样移动插入标记
private void listView1_DragOver(object sender, DragEventArgs e)
{
    //获得鼠标坐标
    Point point = listView1.PointToClient(new Point(e.X, e.Y));
    //返回离鼠标最近的项目的索引
    int index = listView1.InsertionMark.NearestIndex(point);
    //确定光标不在拖曳项目上
    if (index > -1)
    {
        //获取 listView1 控件中所选项的范围
        Rectangle itemBounds = listView1.GetItemRect(index);
        //如果鼠标点在 listView1 控件的被插入项的右半部分
        if (point.X > itemBounds.Left + (itemBounds.Width / 2))
        {
            //则在这个被插入项的后面出现插入标记
            listView1.InsertionMark.AppearsAfterItem = true;
        }
        else
        {
            listView1.InsertionMark.AppearsAfterItem = false;
        }
    }
    listView1.InsertionMark.Index = index;
}

//当鼠标离开控件时移除插入标记
private void listView1_DragLeave(object sender, EventArgs e)
{
    //则不显示插入标记
    listView1.InsertionMark.Index = -1;
}
//将项目移到插入标记所在的位置

```



```

private void listView1_DragDrop(object sender, DragEventArgs e)
{
    //返回插入标记的索引值
    int index = listView1.InsertionMark.Index;
    //如果插入标记不可见，则退出
    if (index == -1)
    {
        return;
    }
    //如果插入标记在项目的右面，使目标索引值加 1
    if (listView1.InsertionMark.AppearsAfterItem)
    {
        index ++;
    }

    //返回拖曳项
    ListViewItem item = (ListViewItem)e.Data.GetData(typeof(ListViewItem));
    //在目标索引位置插入一个拖曳项目的副本
    listView1.Items.Insert(targetIndex, (ListViewItem)item.Clone());
    //移除拖曳项目的原文件
    listView1.Items.Remove(item);
}

//对 ListView 里的各项根据索引进行排序
private class ListViewIndexComparer : System.Collections.IComparer
{
    public int Compare(object x, object y)
    {
        return ((ListViewItem)x).Index - ((ListViewItem)y).Index;
    }
}

```

对程序进行编译并运行后，得到的结果如图 14.55 所示。选中 ListView 控件中的最后一项进行拖曳，如图 14.56 所示，可以看到控件中所选项的目标位置处会出现一个插入标记，松开鼠标，插入标记消失，所选项出现在插入标记所指的位置上，如图 14.57 所示。



图 14.55 原图标视图

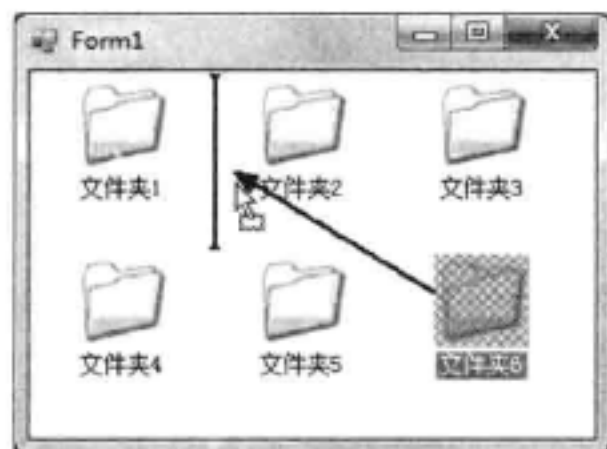


图 14.56 拖动图标




图 14.57 拖放后的效果

14.4.9 在 ListView 中进行搜索

当在 ListView 控件中存在大量列表项的时候，用户会希望其能提供搜索功能。ListView 控件可以通过两种方式完成此项功能：文本匹配和位置搜索。

当使用文本匹配方式时，需要使用到 `FindItemWithText()` 方法。通过此方法可以对 `ListView` 控件上的各项执行文本匹配搜索。下面的代码示例演示如何由用户输入文本在列表中查找项。

```
private ListView listView1 = new ListView();
private TextBox textBox1 = new TextBox();
//当 textBox1 的文本发生变化时
private void textBox1_TextChanged(object sender, EventArgs e)
{
    //调用 FindItemWithText 方法查找 searchBox 里的文本所指示的文本
    ListViewItem listViewItem1 = listView1.FindItemWithText(textBox1.Text,
        false, 0, true);
    //如果查找到了目标项
    if (listViewItem1 != null)
    {
        //则将此项移到 listView1 的最顶端
        listView1.TopItem = listViewItem1;
    }
}
```

 说明：`FindItemWithText()` 方法是 .NET Framework 中已定义好的方法，如果用户希望按照自己的规则对项进行查找，可重写此方法。

14.5 TreeView 控件

`TreeView` 控件用来显示信息的分级视图，如同 Windows 里的资源管理器的目录。`TreeView` 控件一般用来显示文件和目录结构、文档中的类层次、索引中的层次和其他具有分层目录结构的信息。

14.5.1 用 TreeView 控件显示分层数据

`TreeView` 控件在 .NET 中命名空间为 `System.Windows.Forms`。在 Windows 应用程序编程中，`TreeView` 控件可以为用户显示节点层次结构，就像在 Windows 操作系统的 Windows 资源管理器功能的左窗格中，显示文件和文件夹一样，如图 14.58 所示。

树视图中的各个节点可能还包含其他节点，称为“子节点”。




 说明：Windows 应用程序的使用者可以通过展开或折叠这些节点的方式显示父节点或包含子节点的节点。

图 14.58 Windows 资源管理器左窗格

`TreeView` 控件在 Visual Studio 的工具箱中的图示如图 14.59 所示，将 `TreeView` 控件拖

曳到窗体并选定后，如图 14.60 所示。下一步，可在 Visual Studio 的属性窗体中对 TreeView 的属性进行设置，使其达到应用程序的具体要求。

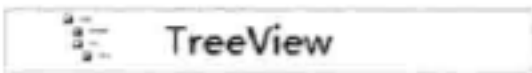


图 14.59 TreeView 控件



图 14.60 TreeView 控件

14.5.2 添加节点到 TreeView

对树添加与删除节点，主要通过对树的属性进行设置和通过代码完成这两种方法。首先就通过属性设置对 TreeView 控件添加节点进行讨论。在 TreeView 控件的属性窗体中选择 Nodes 属性如图 14.61 所示。



图 14.61 TreeView 控件属性设置

点击右侧的…按钮，则会打开“TreeNode 编辑器”窗体，如图 14.62 所示。如果 TreeView 控件的节点集合为空，则先向 TreeView 控件“添加根”，然后根据层级结构添加相应的子节点或根节点。在“TreeNode 编辑器”的右侧对添加的节点进行属性设置。

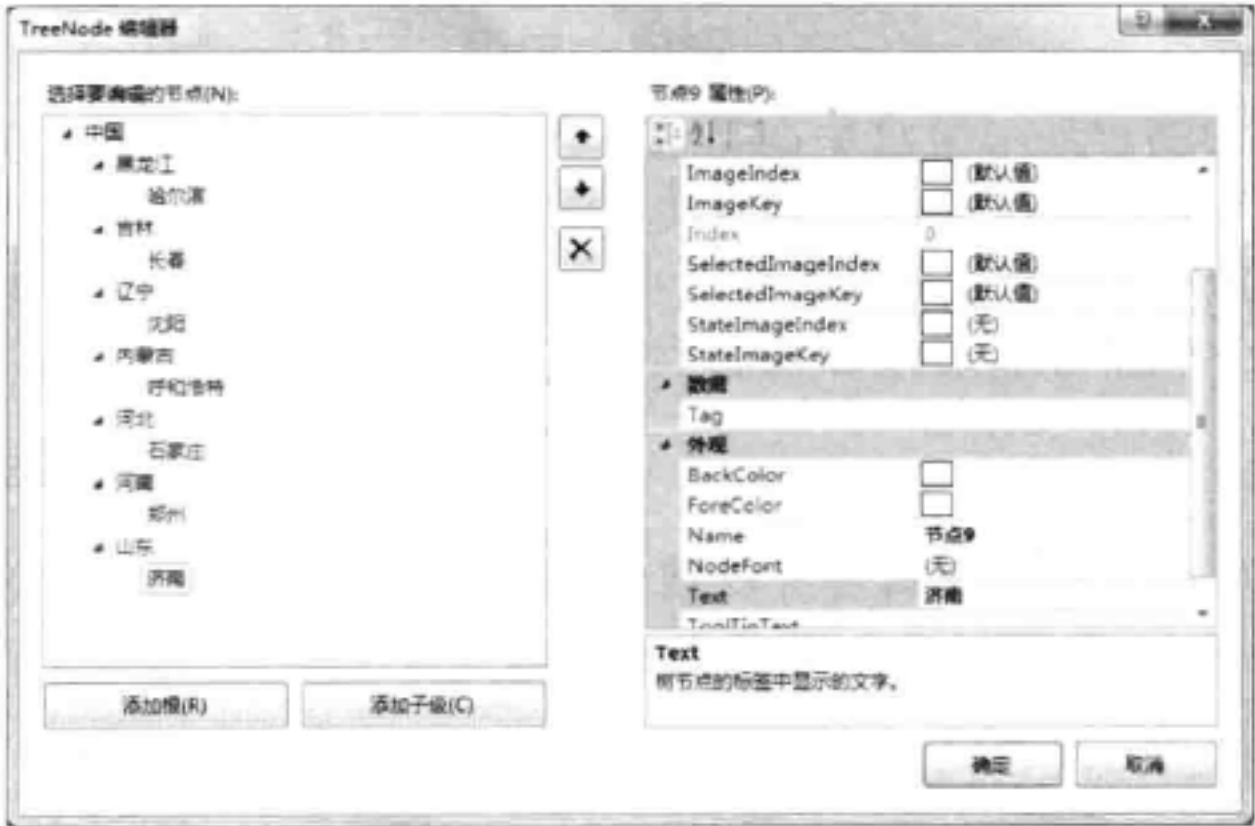



图 14.62 TreeNode 编辑器

 **技巧:** 可以按照一定的层级结构对 TreeView 控件添加节点。


首先, 向 TreeView 控件添加了一个“中国”根节点, 然后在根节点的下面依次加入了“黑龙江”、“吉林”、“沈阳”、“内蒙古”、“河北”、“河南”子节点, 最后在各个省名的节点下, 添加了各省相应的省会作为第 3 层子节点。单击“确定”按钮完成 TreeNode 编辑。运行结果如图 14.63 所示。

也可以使用编程方式向 TreeView 控件添加节点, 所涉及的方法与本章前几节中所讲述的向控件添加子项的方法基本相同, 实现前面在 TreeNode 编辑器中所实现的功能, 具体代码如下:



图 14.63 运行结果

```
private void InitializeTreeView()
{
    //停止绘制 treeView1
    treeView1.BeginUpdate();
    //向 treeView1 添加根节点“中国”
    treeView1.Nodes.Add("中国");
    //向 treeView1 的根节点“中国”添加子节点“黑龙江”
    treeView1.Nodes[0].Nodes.Add("黑龙江");
    //向节点“黑龙江”添加子节点“哈尔滨”
    treeView1.Nodes[0].Nodes[0].Nodes.Add("哈尔滨");
    //向 treeView1 的根节点“中国”添加子节点“吉林”
    treeView1.Nodes[1].Nodes.Add("吉林");
    //向节点“吉林”添加子节点“长春”
    treeView1.Nodes[1].Nodes[0].Nodes.Add("长春");
    //向 treeView1 的根节点“中国”添加子节点“内蒙古”
    treeView1.Nodes[2].Nodes.Add("内蒙古");
    //向节点“内蒙古”添加子节点“呼和浩特”
    treeView1.Nodes[2].Nodes[0].Nodes.Add("呼和浩特");
    //向 treeView1 的根节点“中国”添加子节点“河北”
    treeView1.Nodes[3].Nodes.Add("河北");
    //向节点“河北”添加子节点“石家庄”
    treeView1.Nodes[3].Nodes[0].Nodes.Add("石家庄");
    //向 treeView1 的根节点“中国”添加子节点“河南”
    treeView1.Nodes[4].Nodes.Add("河南");
    //向节点“河南”添加子节点“郑州”
    treeView1.Nodes[4].Nodes[0].Nodes.Add("郑州");
    //向 treeView1 的根节点“中国”添加子节点“山东”
    treeView1.Nodes[5].Nodes.Add("山东");
    //向节点“山东”添加子节点“济南”
    treeView1.Nodes[5].Nodes[0].Nodes.Add("济南");
    //启动绘制 treeView1
    treeView1.EndUpdate();
}
```

 **技巧：**在使用代码向 TreeView 控件添加节点时，也应该按照一定的层级结构进行。

编译运行程序，程序运行结果如图 14.63 所示。

对于如图 14.63 所示的 TreeView 控件运行结果，用户并不能对其进行多选设置，这时候，TreeView 控件通常用于浏览资源或数据，并可以通过选择数据中的某一个节点，而在其他控件中显示其相应的信息。如图 14.64 所示的示例中，在窗体左侧的树型浏览窗体中选择不同的节点，则在窗体右侧的 ListBox 控件中将显示相应节点文件夹中的内容。

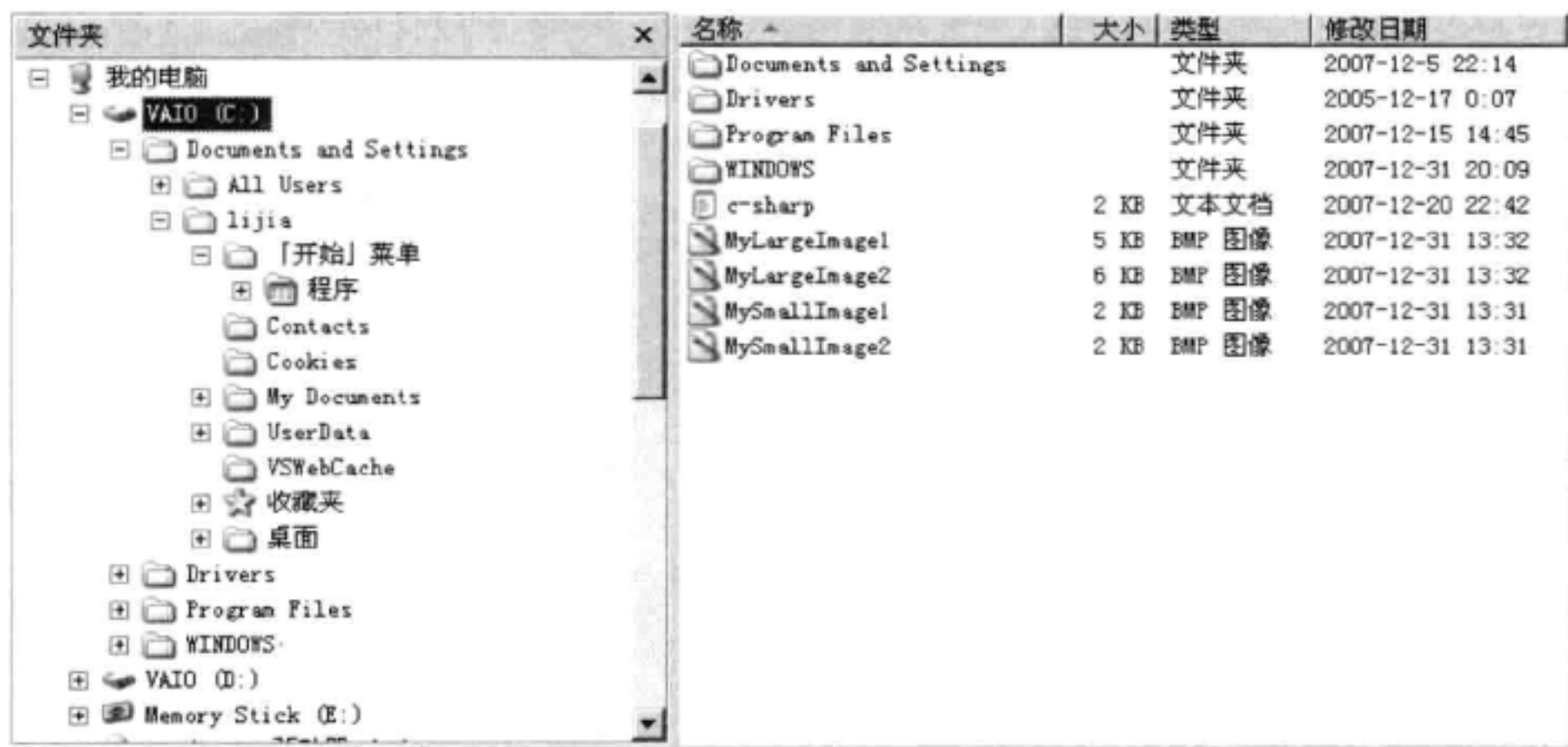



图 14.64 TreeView 控件使用示例

 **说明：**在实际开发应用中，TreeView 控件中的节点都是从数据库或其他文件中读入后自动生成的。

14.5.3 在节点前显示复选框

有很多时候，TreeView 控件不仅仅是用来浏览数据和资源，也可用来让用户在所浏览的数据上进行选择。在通常的应用程序需求中，用户希望可以对 TreeView 控件进行多项选择，这时，可以通过设置 TreeView 控件的 CheckBoxes 属性来完成。CheckBoxes 属性值是个 boolean 类型的变量，当其值为 true 时，TreeView 控件的每个节点旁，将显示一个复选框，如图 14.65 所示。

另外，开发人员也可以通过设置 StateImageList 属性，更改 TreeView 控件外观。当 StateImageList 属性值不为空时，且同时 TreeView 控件的 CheckBoxes 属性为 true，则在 TreeView 控件的每个节点旁，将显示 StateImageList 属性所指示的 ImageList（为 ImageList 给 TreeView 设置图片的操作参见 14.5.5 节）组件中的图片。当节点被选中时，节点前的图标将显示为 StateImageList 属性所指示的 ImageList 组件中的图片里 ImageIndex 为 1 的图片。当节点未被选中时，节点前的图标将显示为 StateImageList 属性所指示的 ImageList 组件中的图片里 ImageIndex 为 0 的图片，如图 14.66 所示。



图 14.65 运行外观



图 14.66 运行外观

在对 TreeView 控件中的各节点进行选择时，将触发 BeforeSelect 事件和 AfterSelect 事件，并可以通过控件的 SelectedNode 属性获取用户选中的节点。BeforeSelect 事件在选定某节点前发生，而 AfterSelect 事件则在选定某节点后发生。

在下面的代码中，将在用户选定了某节点后，将弹出一个消息对话框，在对话框中显示用户选定节点的文本内容。

```
//在对树节点进行选择后引发
private void treeView1_AfterSelect(System.Object sender, TreeViewEventArgs e)
{
    //弹出消息对话框，显示所选节点的内容
    MessageBox.Show(e.Node.Text);
}
```

同样地，下面的代码将通过对 TreeView 控件的 SelectedNode 属性进行访问，实现同样的功能。

```
private void TreeView1_AfterSelect(System.Object sender, TreeViewEventArgs e)
{
    //弹出消息对话框，显示所选节点的内容
    MessageBox.Show(treeView1.SelectedNode.Text);
}
```

在对 TreeView 控件中的各节点进行选中操作时，将触发 BeforeCheck 事件和 AfterCheck 事件。BeforeCheck 事件在选中某节点前发生，而 AfterCheck 事件则在选中某节点后发生。

在下面的示例代码中，将通过对 AfterCheck 事件的处理，使用户在选中树视图中的某一节点时，程序将同时改变它的所有子节点的状态。即当用户选中某一节点时，则此节点的所有子节点也将被选中，而当用户取消对某节点的选中时，则此节点的所有子节点也将被取消选中。

```
//递归更新所有的子节点
private void RefreshChildNode(TreeNode treeNode, bool checked)
{
    //遍历选中节点的所有子节点
    foreach(TreeNode node in treeNode.Nodes)
    {
        node.Checked = checked;
        //遍历此子节点的所有子节点
        if(node.Nodes.Count > 0)
        {

```

```

        //如果现在操作的节点有子节点, 则对当前节点递归调用 AlterChildNode 方法
        RefreshChildNode(node, checked);
    }
}

//注意: 此处只能使用 AfterCheck 事件, 不能使用 BeforeCheck 事件
//当一个根节点的 checked 属性发生变化时, 将它的子节点的 checked 属性设置为与它相同
private void node_AfterCheck(object sender, TreeViewEventArgs e)
{
    //只有当用户改变节点的 checked 属性时, 代码才会被执行
    if(e.Action != TreeViewAction.Unknown)
    {
        //如果选中节点的子节点数不为 0
        if(e.Node.Nodes.Count > 0)
        {
            //调用 AlterChildNode 方法
            //传递当前改变 Checked 属性的树节点和改变后的 Checked 属性值
            RefreshChildNode(e.Node, e.Node.Checked);
        }
    }
}


```

14.5.4 选中节点的常用约定

在实际应用中, 对于 TreeView 控件中的节点状态的更改将不仅限于上面的程序所完成的递归更改子节点的状态。在 Windows 应用程序开发中, 对于 TreeView 控件的使用, 通常有以下几点要求:

- ☐ 当使用者选中某父节点时, 此父节点的全部子节点全部同时选定。
- ☐ 当使用者选中某子节点时, 则此节点的父节点也相应的被选中。
- ☐ 当使用者取消某一节点的选中时, 此节点的所有子节点全部取消选中。
- ☐ 当使用者取消了某父节点的最后一个子节点的选中时, 则此父节点也被取消选中。

对于以上功能的实现, 同样也需要使用到递归方法, 读者可以在以后的开发过程中仔细体会。

说明: TreeView 控件由于使用的广泛性和复杂性, 在使用时通常需要进行大量的二次开发。

14.5.5 用 ImageList 为 TreeView 提供图标

前面已经讲到, 用户可以通过对 TreeView 控件的每个节点前的复选框进行选中来实现树控件中节点的多选, 也可以通过下面的代码来递归遍历树中的节点, 并读取它们的 checked 属性来获取用户当前所选中的项。

```

private void GetCheckedNodes(TreeNode treeNode, ArrayList checkedNodes)
{

```




```

//判断此节点是否选中，如选中则将其添加到列表中
if (treeNode.Checked)
{
    checkedNodes.add(treeNode);
}
//递归获取每个节点的选中的子节点
foreach (TreeNode treeNode1 in treeNode.Nodes)
{
    GetCheckedNodes(treeNode1, checkedNodes);
}
}

```

在本节开始部分的图 14.58 中，可以看到当 TreeView 控件是用于显示数据时，在 TreeView 控件的各节点前可以设置图标使用户更直观地了解各节点所处的状态。对于图标的设置，可以对 TreeView 控件的 ImageList 属性进行设置，设置完成后，TreeView 控件中的各节点将显示出在 ImageList 属性中 ImageIndex 属性为 0 的图片。

 **说明：**当节点被选中时，节点前的图标将显示为 StateImageList 属性所指示的 ImageList 组件中的图片里 ImageIndex 为 1 的图片。当节点未被选中时，节点前的图标将显示为 StateImageList 属性所指示的 ImageList 组件中的图片里 ImageIndex 为 0 的图片。

在下面的代码示例中，将通过对每个节点的 ImageIndex 属性进行设置，使树视图中各节点前的图片的状态，根据其展开或折叠的状态变化而改变。

```

//在 treeView1 被折叠时发生
private void treeView1_AfterCollapse(object sender, TreeViewEventArgs e)
{
    e.Node.ImageIndex = 0;
}
//在 treeView1 被展开时发生
private void treeView1_AfterExpand(object sender, TreeViewEventArgs e)
{
    e.Node.ImageIndex = 1;
}

```

对程序进行编译并运行，程序的运行结果如图 14.67 和图 14.68 所示。



图 14.67 运行外观



图 14.68 运行外观

14.5.6 展开与折叠 TreeView 的节点

在上节的代码示例中，用到了两个 TreeView 控件的事件，分别是 AfterCollapse 事件

和 AfterExpand 事件。AfterCollapse 事件将在 TreeView 控件中的某节点的子节点被折叠后发生, 而 AfterExpand 事件将在 TreeView 控件中的某节点的子节点被展开后发生。相应地, BeforeCollapse 事件将在 TreeView 控件中的某节点的子节点被折叠前发生, 而 BeforeExpand 事件将在 TreeView 控件中的某节点的子节点被展开前发生。

在下面的示例代码中, 将通过添加 TreeView 控件的 BeforeCollapse 事件处理程序, 使程序可以自动判断程序中选中状态的节点, 并取消此节点的折叠操作, 使 TreeView 控件中所有的选中节点都是可见的。

```
private void button1_Click(object sender, EventArgs e)
{
    //停止 TreeView 控件的自我绘制
    treeView1.BeginUpdate();
    //展开树中的所有节点
    treeView1.ExpandAll();
    //注册 treeView1_BeforeCollapse 事件处理函数
    treeView1.BeforeCollapse += new TreeViewCancelEventHandler(treeView1_
    BeforeCollapse);
    //折叠树中的所有节点
    treeView1.CollapseAll();
    //取消注册 treeView1_BeforeCollapse 事件处理函数
    treeView1.BeforeCollapse -= new TreeViewCancelEventHandler(treeView1_
    BeforeCollapse);
    //停止 TreeView 控件的自我绘制
    treeView1.EndUpdate();
}
//在折叠 treeView1 时发生
private void treeView1_BeforeCollapse(object sender, TreeViewCancel
EventArgs e)
{
    //在节点被折叠前, 调用 IfChildNodeChecked() 方法, 检查此节点是否有子节点被选中
    bool ifChildNodeChecked = IfChildNodeChecked(e.Node)
    //如果此节点有子节点被选中, 则取消折叠此节点
    if (ifChildNodeChecked)
    {
        e.Cancel = true;
    }
}
//检查传入的节点是否有子节点被选中
private bool IfChildNodeChecked(TreeNode node)
{
    //检查此节点是否含有子节点
    if (node.Nodes.Count == 0)
    {
        return false;
    }
    //若含有子节点, 则检查子节点中是否有选中的节点
    else
    {
        //遍历选中节点的所有此节点
        foreach (TreeNode childNode in node.Nodes)
```




```

{
    //如果此节点被选中
    if (childNodes.Checked)
    {
        return true;
    }
    //递归调用 IfChildNodeChecked 方法, 检查子节点的子节点中, 是否含有选中的节点
    //注意: 这一步是必要且必须的
    //因为 CollapseAll() 方法是递归的对每一个节点执行 Collapse() 方法
    //而每一个 Collapse() 方法只是对当前节点进行折叠
    //并不会递归的折叠此节点的所有子节点
    if (IfChildNodeChecked(childNode))
    {
        return true;
    }
}
return false;
}
}

```

14.5.7 允许编辑 TreeView 的节点文本

在 TreeView 控件中, 有时候用户希望可以对树节点中的文本进行编辑, 为了满足这个需求, 需要将 TreeView 控件的 LabelEdit 属性设置为 true。

 **注意:** 只是将 LabelEdit 属性设置为 true, 并不能使 TreeView 控件中的节点可编辑。

在下面的程序中, 将具体实现一个树节点的文本重新编辑功能, 代码如下:

```

private void button1_Click(object sender, EventArgs e)
{
    //获取树视图中选中的节点
    TreeNode selectedNode = treeView1.SelectedNode;
    //判断树中是否有选中节点
    if (selectedNode != null)
    {
        //判定选中节点是否为根节点
        if (selectedNode.Parent != null)
        {
            //使选中节点进入编辑状态
            treeView1.LabelEdit = true;
            selectedNode.BeginEdit();
        }
        else
        {
            //如果选中的是根节点, 则弹出提示
            MessageBox.Show("选中的是根节点.");
        }
    }
}
else
{


```

```

        //如果没有树节点被选中, 则弹出提示
        MessageBox.Show("没有树节点被选中.");
    }
}
//当对 treeView1 中的某节点进行文本编辑后
private void treeView1_AfterLabelEdit(object sender, NodeLabelEdit-
EventArgs e)
{
    if (e.Label != null && e.Label.Length > 0)
    {
        //判断输入的文本是否含有违法字符
        if (e.Label.IndexOfAny(new char[]{'@', '.', ',', '!', '\\'}) == -1)
        {
            //停止编辑
            e.Node.EndEdit(false);
        }
        else
        {
            //取消文本的编辑, 将节点置于可编辑状态
            e.CancelEdit = true;
            MessageBox.Show("无效的树节点标签.\n");
            e.Node.BeginEdit();
        }
    }
    else
    {
        //取消文本的编辑, 将节点置于可编辑状态
        e.CancelEdit = true;
        MessageBox.Show("无效的树节点标签\r\n 标签项不能为空");
        e.Node.BeginEdit();
    }
    this.treeView1.LabelEdit = false;
}

```

在上面的例子中, 使用到了 AfterLabelEdit 事件, 这个事件发生在对文本编辑完成后。与树节点的文本编辑相关的还有 BeforeLabelEdit 事件, 此事件主要发生在文本编辑正要开始的时候。

 **技巧:** 可如上例中所示, 使用 BeforeLabelEdit 事件来检验所选节点是否合理, 使用 AfterLabelEdit 事件来检验所输入的文本是否都符合标准。

14.5.8 让 TreeView 支持拖曳操作

在使用 Windows 资源管理器时, 用户通常会在资源管理器的左窗格中对任意文件夹进行拖曳, 从而实现文件夹之间的复制或移动。在 TreeView 控件中, 同样也可以通过添加相应的事件处理程序来实现上述的功能。

在下面的程序中, 可以将树视图的非根节点通过拖曳操作移动或复制到树视图的另一个节点上, 使被移动节点作为所选节点的子节点。当对 TreeView 控件中的某节点进行拖曳

时，被拖动的节点将被移动到目标节点。

```
public class Form1 : Form
{
    private TreeView treeView1;
    //Form1()的构造函数
    public Form1()
    {
        treeView1 = new TreeView();
        //初始化 treeView1.
        treeView1.AllowDrop = true;
        treeView1.Dock = DockStyle.Fill;
        //向 treeView1 添加节点
        TreeNode node=new TreeNode();
        for (int x = 1; x < 4; x++)
        {
            //向 treeView1 添加根节点
            node = treeView1.Nodes.Add("节点"+x.ToString());
            for (int y = 1; y < 5; y++)
            {
                //向之前添加的节点添加一个子节点
                node = node.Nodes.Add("节点"+x.ToString()+"."+y.ToString());
            }
        }
        //展开所有节点
        treeView1.ExpandAll();
        //注册拖曳事件
        treeView1.ItemDrag += new ItemDragEventHandler(treeView1_ItemDrag);
        treeView1.DragEnter += new DragEventHandler(treeView1_DragEnter);
        treeView1.DragOver += new DragEventHandler(treeView1_DragOver);
        treeView1.DragDrop += new DragEventHandler(treeView1_DragDrop);
        //初始化窗体
        this.ClientSize = new Size(300, 400);
        this.Controls.Add(treeView1);
        this.ResumeLayout(false);
    }

    //对 treeView1 的节点进行拖曳时发生
    private void treeView1_ItemDrag(object sender, ItemDragEventArgs e)
    {
        //当对某节点进行拖曳时，移动拖曳的节点
        if (e.Button == MouseButtons.Right)
        {
            DoDragDrop(e.Item, DragDropEffects.Move);
        }
    }

    //当鼠标进入 treeView1 时发生
    private void treeView1_DragEnter(object sender, DragEventArgs e)
    {
        e.Effect = e.AllowedEffect;
    }

    //鼠标光标在控件的界限内移动时被引发
    private void treeView1_DragOver(object sender, DragEventArgs e)
```



```

{
    //获得鼠标的坐标
    Point point = treeView1.PointToClient(new Point(e.X, e.Y));
    //按鼠标所指示的位置选择节点
    treeView1.SelectedNode = treeView1.GetNodeAt(point);
}
//在完成拖放操作时被引发
private void treeView1_DragDrop(object sender, DragEventArgs e)
{
    //获得释放鼠标位置的坐标
    Point point = treeView1.PointToClient(new Point(e.X, e.Y));
    //获得在鼠标释放处的节点
    TreeNode node1 = treeView1.GetNodeAt(point);
    //获得当前拖曳的节点
    TreeNode node2 = (TreeNode)e.Data.GetData(typeof(TreeNode));
    //确定在释放位置的节点既不是拖曳的节点也不是拖曳节点的子节点
    if (!node2.Equals(node1) && !HasChildNode(node2, node1))
    {
        //如果是一个移动操作,将当前位置的节点移动到鼠标释放的位置
        if (e.Effect == DragDropEffects.Move)
        {
            node2.Remove();
            node1.Nodes.Add(node2);
        }
        //如果是一个复制操作,复制拖曳的节点,并将它添加到鼠标释放的位置
        else if (e.Effect == DragDropEffects.Copy)
        {
            node1.Nodes.Add((TreeNode)node2.Clone());
        }
        //在释放节点的位置上展开这个节点
        node1.Expand();
    }
}
//确定 node1 是否为 node2 的父节点
private bool HasChildNode(TreeNode node1, TreeNode node2)
{
    //判断第二个节点是否是父节点
    if (node2.Parent == null)
        return false;
    if (node2.Parent.Equals(node1))
        return true;
    //递归的调用 IfParentNode 方法
    //判断 node1 是否为 node2 的父节点
    return HasChildNode(node1, node2.Parent);
}
}

```

编译并运行上面的程序,得到的结果如图 14.69 所示。选中 TreeView 控件中的一个子节点进行拖曳,拖曳至用户想要添加的目标节点处,如图 14.70 所示,可以看到控件中所选项的目标节点将会被涂蓝,松开鼠标,所选子节点将成为目标节点的子节点,如图 14.71 所示。

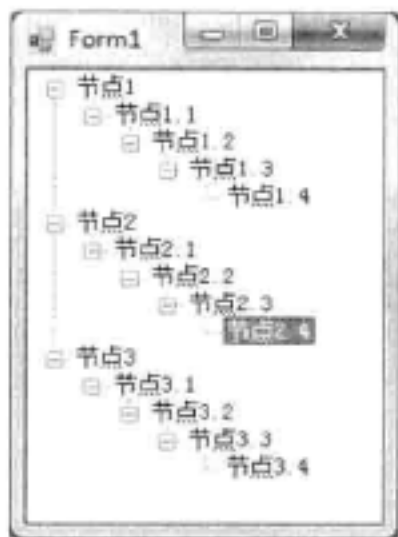


图 14.69 运行外观

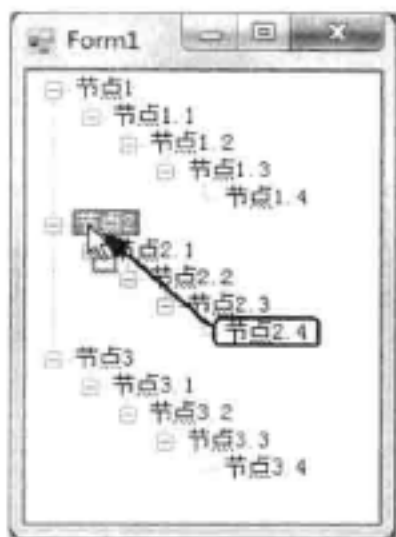


图 14.70 运行外观

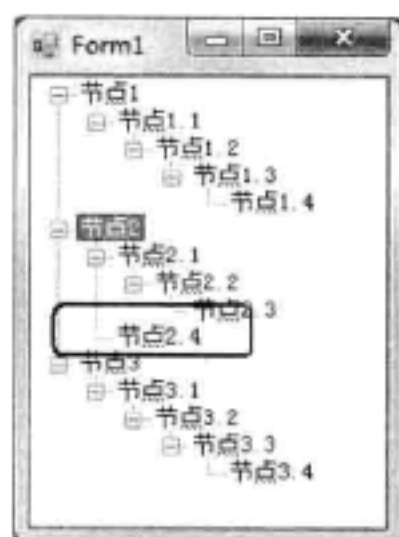


图 14.71 运行外观

注意：由于拖曳操作经常会使树结构变得复杂，所以除非用户要求，否则不必向 TreeView 控件添加拖曳操作。

14.6 本章总结

在本章中，介绍了 .NET 控件所提供的常用列表选择控件，这些控件分别为列表框 (ListBox)、复选列表框 (ComboBox)、下拉列表框 (CheckedListBox)、列表视图 (ListView) 和树视图 (TreeView) 5 种。

ListBox 控件用于显示一个项列表，用户可从中选择一项或多项。

ComboBox 控件用于在下拉组合框中显示数据。

CheckedListBox 控件几乎具有 ListBox 控件的所有功能，同时 CheckedListBox 控件还可以在列表项的旁边显示复选标记。

ListView 控件可以为用户显示带图标项的列表。

TreeView 控件可以为用户显示节点层次结构。

读者在进行 Windows 应用程序开发时，可根据用户的实际需求选择合适的列表选择控件，并通过对控件的属性、事件和方法的使用，来完成特定的功能。

14.7 实战练习

1. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，要求使用 CheckedListBox 控件设计一个窗体，在窗体左侧显示餐厅的菜单列表，当用户选中一些菜单并单击“添加”按钮后，即可将选中的菜品名称添加到右侧的列表框。还需要设计一个添加菜品的功能，当用户在文本框中输入新菜品名称后，将其添加到左侧菜单列表中。

2. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，使用 ListView 控件模拟显示 Windows 的控制面板，可按大图标、小图标、列表、详细资料等方式显示控制面板各项的内容。

3. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，使用 TreeView 控件显示餐厅的菜单，菜单分为两大类：中餐、西餐。其中中餐又分为凉菜、热菜、汤等几种小类，在各小类中再添加详细的菜品名称。西餐又分为法式、英式、意式、俄式、美式等几种小类，在各小类中再添加详细的菜品名称。

第 15 章 数据显示控件

在.NET 中,数据显示控件主要有 DataGrid 和 DataGridView 两种。其中,DataGridView 控件是.NET 2.0 中的一个添加的控件,并且在.NET 3.0 和.NET 4.0 中被继续沿用。DataGridView 控件主要是针对.NET 1.x 中的相对可用性较差的 DataGrid 控件开发的,也就是说,DataGridView 控件是 DataGrid 控件的发展与延续。它不但具有 DataGrid 控件的全部功能,同时与 DataGrid 控件相比,具有更高的稳定性、可开发性和用户友好性。因此在本章中,将重点针对 DataGridView 控件的功能进行说明,对该控件在应用程序中的应用和编程中的具体方法进行阐述,并建议读者在今后的开发过程中,也尽量选用该控件。

15.1 DataGridView 控件以表格显示数据

15.1.1 DataGridView 有哪些功能

DataGridView 控件在.NET 的命名空间是 System.Windows.Forms。在 Windows 应用程序编程中,该控件是以表格形式来显示数据的一种控件。该控件在 Visual Studio 的工具箱中的图示如图 15.1 所示。将其控件拖曳到窗体并选定后,如图 15.2 所示。



图 15.1 DataGridView 控件

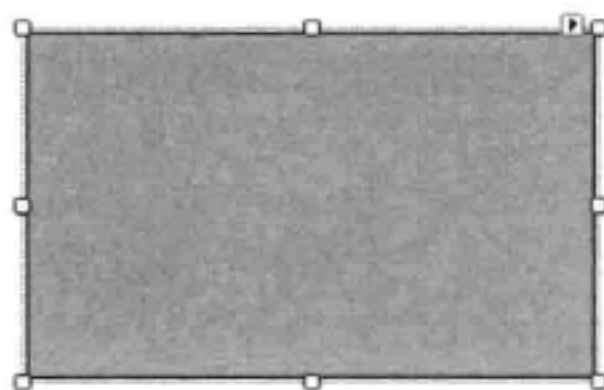



图 15.2 DataGridView 控件

可根据 DataGridView 控件的属性进行设置,定制符合用户需求的该控件的外观和简单行为。

DataGridView 控件为开发人员提供了在数据显示时可用到的大量的默认功能,通过使用这些功能,可以使开发过程更加简单、高效。

 **说明:** 开发人员可以通过对属性的定制和相关方法的引用轻松地使用这些已经定义好的功能。

这些功能主要有:

□ 当 DataGridView 控件所要显示的数据超过 DataGridView 控件的显示范围时,将自

动显示水平滚动条和垂直滚动条。

- ☐ 当显示水平滚动条和垂直滚动条时，始终保持行标头和列标头可见。
- ☐ 当用户需要时，可对 DataGridView 控件实行添加新数据行操作，并可以同步更新到相应的数据源。
- ☐ 当用户需要时，可对 DataGridView 控件实行编辑现有数据行操作，并可以同步更新到相应的数据源。
- ☐ 当用户需要时可对 DataGridView 控件实行删除现有数据行操作，并可以同步更新到相应的数据源。
- ☐ 用户可以调整显示时控件中各列的宽度。
- ☐ 用户可以选中控件中的整行或整列。
- ☐ 用户可以按照列中数据的基础排序法对数据进行排序。
- ☐ DataGridView 控件绑定到某数据源时，该控件可以为数据源中的列自动创建相对应的列；将数据源中各列的名称自动作为列标头；用数据源的内容对该控件绑定进行填充；为数据源中的每个合法数据行创建一行。

通过以上的功能描述可以看出，DataGridView 控件具有极其强大的功能，能以用户需要的各种形式显示各种数据来源中的各种数据类型。另外，其功能的强大性还体现在以下几点中：

- ☐ 它不但可以显示只读数据，也可以显示可编辑的数据。
- ☐ 它不但可以显示少量的应用程序数据，也可以显示海量的数据库数据。
- ☐ 它不但可以通过滑动条在一页中显示全部的数据，也可以通过二次开发实现数据的分页显示。
- ☐ 它不但可以显示文本型数据，也可以显示图片型数据。
- ☐ 它不但可以用 TextBox 格式设置每个单元格，也可以用 ComboBox 格式设置每个单元格。
- ☐ 它不但可以用嵌套 DataGridView 控件的形式显示具有层次结构的数据，也可以用 TreeView 控件的形式显示具有层次结构的控件。


 **说明：**开发人员可以通过各种方式对 DataGridView 控件进行扩展，将系统需求中自定义的行为附加到该控件中，实现该控件的二次开发，定制符合用户需求的 DataGridView 控件。

例如，通过对 DataGridView 控件中的相关类 DataGridViewCell 类中的属性进行重新设置，定制出符合用户外观需求的单元格。通过对 DataGridViewCell 类中的有效性检查进行设置，定制出符合用户业务需求的 DataGridViewCell 类。通过对 DataGridView 控件中的排序方法进行重写，则可以使 DataGridView 控件具有符合用户业务规则的排序方法。可见，DataGridView 控件对于 Windows 应用程序开发具有很高的灵活性。

15.1.2 设置对表格数据的操作

在 Visual Studio 的窗体设计器中，提供了详细的对 DataGridView 控件进行设置的方法。

通过在窗体设计器上进行设置，可以使该控件满足大多数的系统需求。但对于大型系统的开发、复杂度较高的需求，以及对该控件的二次开发等任务，还是建议通过使用编码的方式予以实现。

选中窗体上的 DataGridView 控件，右上角将出现一个向右指示的小箭头。单击此小箭头，将显示出如图 15.3 所示的对话框。通过对此对话框进行编辑，将对该控件的基本任务进行定义。“启用添加”复选框主要用来判定在程序运行时，用户是否可以向该控件输入新的数据行。如果“启用添加”复选框被选中，则在 Windows 应用程序中，将会有一个新的记录行呈现于该控件的底部。双击该行中的每一个单元格，可进入该单元格的编辑模式，实现对新数据行的编辑和添加。

进入该单元格的编辑模式后，此单元格所在的行前的行标签将由一个铅笔图案的图标来表示。而在 DataGridView 控件的底部会出现新的一行数据行，在行标签处以*表示，如图 15.4 所示。



图 15.3 DataGridView 任务




图 15.4 行标记

此特性也可在代码中定义，主要是通过对 AllowUserToAddRows 属性进行设置来实现的。当此属性设置为 true 时，用户可以向该控件添加新的数据行。具体函数代码如下：

```
private void setAllowUserToAddRows ( )
{
    //不允许用户向 dataGridView1 添加行
    this.dataGridView1. AllowUserToAddRows = false;
}
```

此函数可添加到包含 DataGridView 控件的窗体的 Form1_Load 事件处理程序中，或 Visual Studio 自动生成的 InitializeComponent 函数中。

 **说明：**如果 DataGridView 控件是绑定到相应数据源的，则允许用户在此属性以及数据源的 IBindingList.AllowNew 属性均设置为 true 时添加行。

“启用删除”复选框主要用来判定在程序运行时，用户是否可以在 DataGridView 控件中删除现有的数据行。如果“启用删除”复选框被选中，则在 Windows 应用程序运行时，用户可以在选定需要删除的数据行后，按 Del 键实现对选中数据行的删除，同时程序会将这一结果自动更新到 DataGridView 控件相应的单元格中。

此特性也可在代码中定义，主要是通过对 AllowUserToDeleteRows 属性进行设置来实现的。当此属性设置为 true 时，用户可以移除 DataGridView 控件中的现有数据行。具体函数代码如下所示。


```
private void set AllowUserToDeleteRows ( )
{
    //不允许用户对 dataGridView1 删除行
    this.dataGridView1.AllowUserToDeleteRows = false;
}
```

此函数可添加到包含 DataGridView 控件的窗体的 Form1_Load 事件处理程序中，或 Visual Studio 自动生成的 InitializeComponent 函数中。

“启用编辑”复选框主要用来判定在程序运行时，用户是否可以对 DataGridView 控件中的数据行进行编辑。如果“启用编辑”复选框被选中，则在 Windows 应用程序中，用户可以在选定需要编辑的单元格后，通过双击鼠标或按 F2 键进入该单元格的编辑模式。


进入该单元格的编辑模式后，此单元格所在的行前的行标将由一个铅笔图案的图标表示。当用户编辑完成且退出该单元格时，程序会自动同步更新相应的数据源。

此特性也可在代码中定义，主要是通过对 ReadOnly 属性进行设置来实现的。当将此属性设置为 true 时，用户可以对 DataGridView 控件中的数据进行编辑。具体函数代码如下所示。

```
private void setReadOnly ( )
{
    //将 dataGridView1 设为只读
    this.dataGridView1.ReadOnly= true;
}
```

此函数可添加到 DataGridView 控件的窗体的 Form1_Load 事件处理程序中，或 Visual Studio 自动生成的 InitializeComponent 函数中。另外，值得注意的是，如果 DataGridView 控件的 ReadOnly 属性设置为 true，则用户也不能在 DataGridView 控件中添加和删除数据行，即 AllowUserToAddRows 属性与 AllowUserToDeleteRows 属性将被设置为 false。

“启用列重新排序”复选框主要用来判定在程序运行时，用户是否可以对 DataGridView 控件中的数据根据某一列的值进行排序。排序的方法由 .NET 提供，主要规则是对于数字类型的数据以数字大小进行排序，对于字符串类型的数据以字符编码大小进行排序，首先以首字符的编码大小为基准进行排序，当首字符相同时，则以第二个字符为基准进行排序，依此类推。

 **注意：**当数据完全相同时，则以数据起始时在 DataGridView 控件中的位置进行排序。

如果“启用列重新排序”复选框被选中，则在 Windows 应用程序中，用户可以通过单击选定的排列基准列的列标头进行排序。单击后，如果列标头右侧显示的是倒三角形状，则表示此列中的数据以降序法进行排列，并重新排列 DataGridView 控件中的数据行。如果希望选中列的数据呈升序排列，则可再次单击选定的排列基准列的列标头，则此列标头的右侧显示的是正三角形状，列中的数据以升序排列。如图 15.5 所示为 DataGridView 控件中的数据的原始排序方式。如图 15.6 所示的是 DataGridView 控件中的数据的降序排序方式。如图 15.7 所示的是 DataGridView 控件中的数据的升序排序方式。

此特性也可在代码中定义，主要是通过对 AllowUserToOrderColumns 属性进行设置来实现的。当此属性设置为 true 时，用户可以对 DataGridView 控件中的数据按列进行排序。具体函数代码如下：

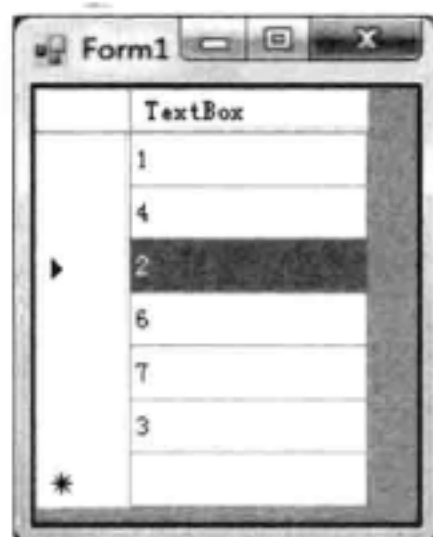


图 15.5 数据原排列方式

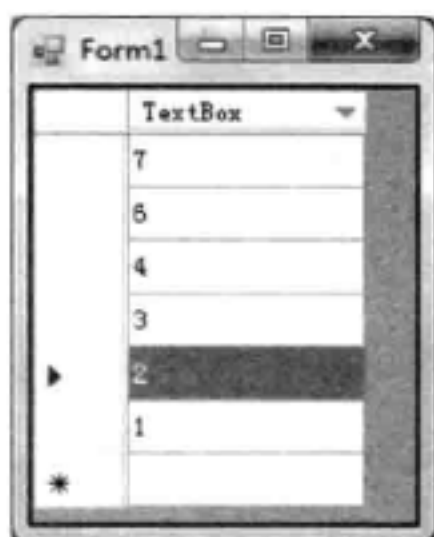


图 15.6 数据降序排列方式

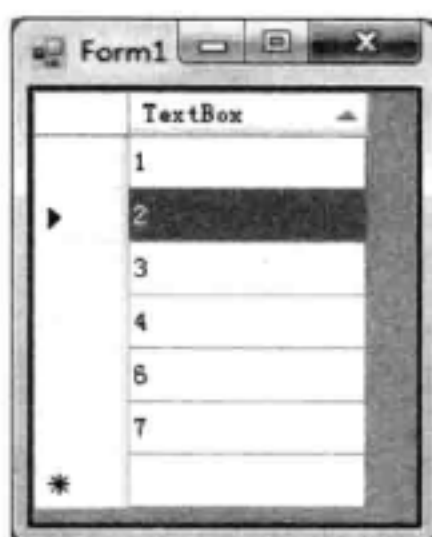


图 15.7 数据升序排列方式

```
private void setAllowUserToOrderColumns ( )
{
    //允许用户对 dataGridView1 按列进行排序
    this.dataGridView1.AllowUserToOrderColumns = true;
}
```

此函数可添加到包含 DataGridView 控件的窗体的 Form1_Load 事件处理程序中或 Visual Studio 自动生成的 InitializeComponent 函数中。

15.1.3 在窗体中如何放置 DataGridView

在图 15.3 中的 DataGridView 任务对话框中，单击最下面的“在父容器中停靠”命令，将把 DataGridView 的 Dock 属性设置为 Fill。即 DataGridView 控件将充满整个 Windows 窗体，而此命令的名称也相应地变为“取消在父容器中停靠”。

当窗体中只有 DataGridView 控件时建议单击“在父容器中停靠”命令，但是在大多数情况下，窗体中还会显示一些其他的控件以表达完整的业务功能。通常含有 DataGridView 控件的窗体如图 15.8 所示。



图 15.8 DataGridView 控件应用举例

从图中看出，在含有 DataGridView 控件的窗体中，该控件虽然不是唯一的控件，却通常是在窗体中所占面积最大、权重相对最高的一个控件。通常含有该控件的窗体功能也都是围绕着该控件展开的。因此在使用该控件时，推荐将 Dock 属性赋予一个值，使程序具有更高的用户友好性。在本例中，该控件的 Dock 值为 DockStyle.Top，也可用以下函数表示：

```
private void setDataGridViewDock ( )
{
    //将 dataGridView1 控件停靠在窗体顶部
    this.dataGridView1.Dock = System.Windows.Forms.DockStyle.Top;
}
```


15.1.4 为 DataGridView 提供数据

在图 15.3 中的 DataGridView 任务对话框中,单击最上面的“选择数据源”旁的 ComboBox 控件的下拉箭头,将打开一个如图 15.9 所示的窗体。在此窗体中将会列举已经添加的数据源,如果其中有此控件需要的数据源则双击选定,DataGridView 控件将会根据数据源数据的列名,自动生成 DataGridView 控件中的列,并完成数据的自动填充。如果如图 15.9 所示没有候选的数据源或没有需要的数据源,则可单击下面的“添加项目数据源”文本标签,弹出“数据源配置向导”对话框,如图 15.10 所示。




图 15.9 DataGridView 控件的数据源



图 15.10 “数据源配置向导”对话框

在数据源配置向导中,可以看到一共有 4 种数据源可供选择,“数据库”数据源是将程序连接到数据库后,针对特定的数据库对象创建的数据集;“服务”数据源是通过一个名为“添加服务引用”的窗体,创建一个服务的连接,并返回需要的应用程序数据;“对象”数据源是指开发人员可以在此选择一个控件,用于以后生成数据绑定控件;“SharePoint”数据源可连接获取 SharePoint 中的数据。

 **注意:** 对于这 4 种不同数据源的使用方法将在后面进行具体的说明。

15.1.5 向 DataGridView 添加列

在图 15.3 的 DataGridView 任务对话框中,单击“添加列...”文本框将弹出一个如图 15.11 所示的窗体。在此窗体中可通过选择每一个添加列的类型来添加相应的列,单击“类型”ComboBox 控件的下拉箭头,将列举出所有的 column 类型。如图 15.12 所示。



图 15.11 “添加列”对话框

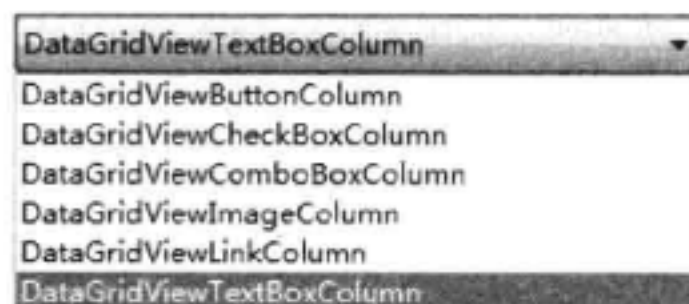


图 15.12 column 类型选择

在图 15.11 所示的添加列对话框下方，有 3 个 CheckBox，分别是可见、只读和冻结。

当“可见”复选框被选中时，指示此列在 DataGridView 控件中可见，此复选框在默认情况下是被选中的。如果此复选框未被选中，则此列在 DataGridView 控件显示时为隐藏状态，不会显示出来。

有时用户会希望仅显示 DataGridView 控件中可用的某些列。例如，向具有管理凭据的用户显示雇员工资列而对其他用户隐藏该列。此外，用户也可能希望将控件绑定到包含许多列的数据源但仅显示其中的某些列。在这种情况下，通常会移除而不是隐藏自己不希望显示的列。在以上情况中 DataGridViewColumn 应设置为不可见的。

说明：也可以在 DataGridViewColumn 的属性窗体中进行设置，DataGridViewColumn 的 Visible 属性值决定是否显示该列。

当在程序运行期间进行判定并确定 DataGridViewColumn 是否可见时，可以以编程方式隐藏列，只需将 DataGridViewColumn.Visible 属性设置为 false。如需要隐藏 DataGridView 控件中的第 1 列，只需要添加如下函数代码即可：

```
private void setColumnVisible ( )
{
    //将 dataGridView1 的第一列设为不可见
    this.dataGridView1.Columns[0].Visible = false;
}
```

使用以上函数将在视图中隐藏第 1 列，但同时在 DataGridView 中保持该列。若要完全移除该列，则需要调用 DataGridViewColumnCollection.Remove() 方法。

另外，若要隐藏的列是在绑定数据源时自动生成的列，则需要在 DataBindingComplete 事件的处理程序中设置此属性。

15.1.6 冻结 DataGridView 中的列

当“冻结”复选框被选中时，则该列会在 DataGridView 控件中被冻结。冻结一列后，


其左侧（在从右到左的字符集中为右侧）的所有列也将被冻结。冻结的列保持不动，而其他所有列可以滚动。

用户在查看 Windows 窗体 DataGridView 控件中显示的数据时，有时需要频繁参考一行或若干行。例如，显示包含多列的用户信息表时，始终显示用户名称而使其他列在可视区域以外滚动会很有用。要实现此行为，可以冻结控件中的用户名所在列。

可以在 DataGridViewColumn 的属性窗体中进行设置，DataGridViewColumn 的属性值 Frozen 决定是否将此列及此列左边所有的列冻结。

也可以以编程方式冻结列，只需将 DataGridViewColumn.Frozen 属性设置为 true。如需要冻结 DataGridView 控件中的第 1 列，只需要添加如下函数代码即可：

```
private void setColumnFrozen ( )
{
    //将 dataGridView1 的第一列冻结
    this.dataGridView1.Columns[0].Frozen = true;
}
```

说明：如果允许对列进行重新排序，则将冻结的列视为一组，以区别于未冻结的列。用户可重新调整冻结和未冻结这两个组中列的位置，但不能将其中一组中的列移动到另一组。某列的 Frozen 属性确定该列在 DataGridView 控件内是否始终可见。

当“只读”复选框被选中时，用户将不能再编辑该列的单元格。在实际程序应用中，并不是所有数据都可编辑。通过对该复选框进行选择，可以指示出用户是否可编辑该列中的单元格。

可以在 DataGridViewColumn 的属性窗体中进行设置，DataGridViewColumn 的属性值 ReadOnly 确定用户是否可编辑该列中的单元格。

也可以以编程方式进行设置，只需将 DataGridViewColumn.ReadOnly 属性设置为 true，则可将此列中的所有单元格设置为只读。具体的实现函数如下所示。

```
private void setColumnReadOnly ( )
{
    //将 dataGridView1 的第一列设为只读
    dataGridView1.Columns[0].ReadOnly = true;
}
```

向 DataGridView 控件添加每种类型的 DataGridViewColumn，可以观察每种类型的 column 的外观特点，如图 15.13 和图 15.14 所示。在这个例子中，将 TextBox 列设置为冻结状态。

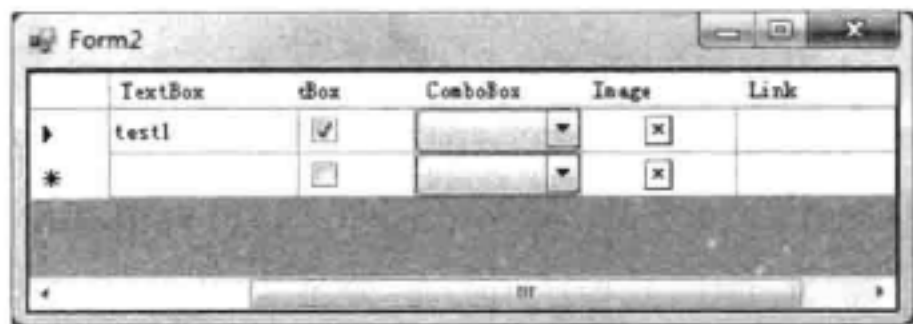


图 15.13 Column 外观比较

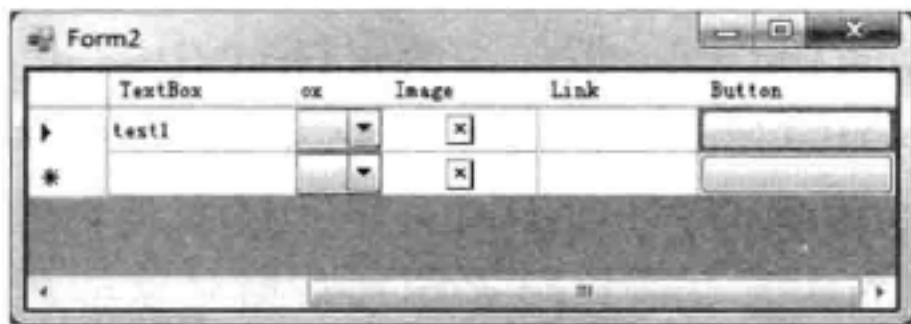


图 15.14 Column 外观比较

15.1.7 编辑 DataGridView 中的列

在图 15.3 的 DataGridView 任务对话框中，单击“编辑列...”文本框将弹出一个如

图 15.15 所示的“编辑列”对话框。在此对话框的左侧是一个 ListBox，在此控件里，列举了 DataGridView 控件中的所有列的名称。可以通过单击 ListBox 控件右侧的箭头按钮来改变各列在 DataGridView 控件中的排列顺序，也可以通过单击 ListBox 控件下方的“添加”按钮向 DataGridView 控件添加新的列，或者单击“移除”按钮删除选定列。对话框右边是一个列表，可以显示选中列的相关属性，每种类型的列所对应的属性也不尽相同，具体属性的设置方法，将在后面的章节中介绍。




图 15.15 “编辑列”对话框

15.2 为 DataGridView 服务的类

DataGridView 控件及为其服务的类为开发人员提供了一个灵活的、高度可配置的系统，通过此控件可以显示和编辑绝大多数的表格数据。这些类全部包含在 System.Windows.Forms 命名空间中，并全部以 DataGridView 前缀命名。其中，主要的 DataGridView 派生类从 DataGridViewElement 派生。

15.2.1 DataGridViewElement 对象模型

DataGridView 控件由 DataGridViewCell 和 DataGridViewBand 两个基本类型对象组成。其中，DataGridViewBand 又分为 DataGridViewColumn 和 DataGridViewRow 这两种类型。DataGridView 控件中的所有单元格都是从 DataGridViewCell 基类派生的。而 DataGridView 控件中的所有行数据对象和列数据对象都是从 DataGridViewBand 基类派生的。

 **说明：**DataGridView 控件可以与多个它的派生类进行互操作，其中，最常用的类为 DataGridViewCell、DataGridViewColumn 和 DataGridViewRow 等 3 种。

DataGridViewElement 类为 DataGridView 控件的元素（即 DataGridViewCell、DataGridViewColumn 和 DataGridViewRow）提供了基类。如图 15.16 显示了 DataGridViewElement

的层次继承结构。

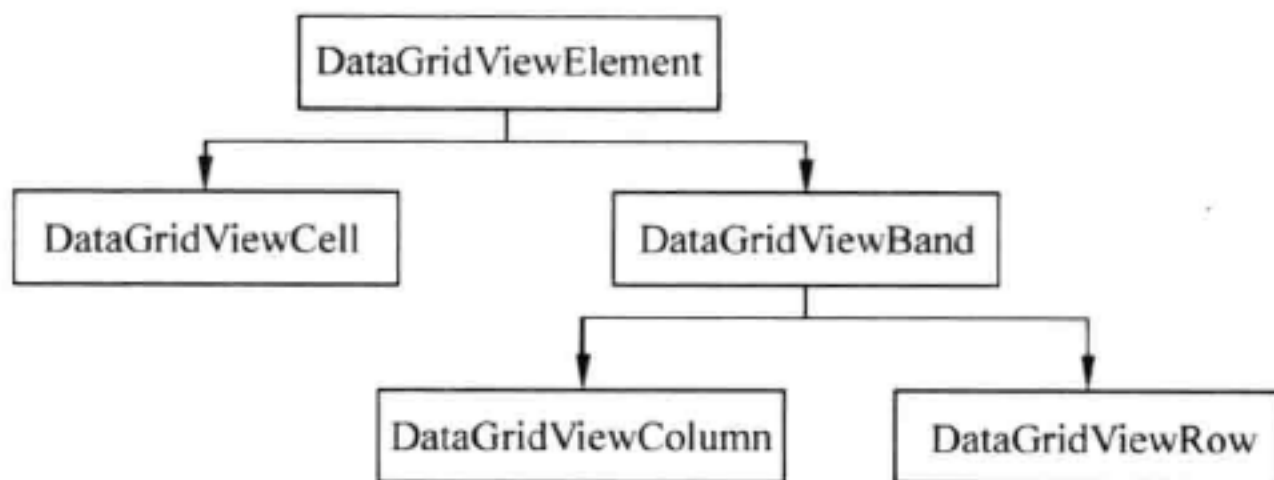


图 15.16 DataGridViewElement 继承层次结构

15.2.2 单元格对象模型 DataGridViewCell

单元格是 DataGridView 控件的最小组成部分，可以说是组成 DataGridView 控件的原子对象，是 DataGridView 控件的基本交互对象。DataGridViewCell 类只是一个抽象基类，所有单元格类型都是从该类派生的。如图 15.17 所示为 DataGridViewCell 继承层次结构。

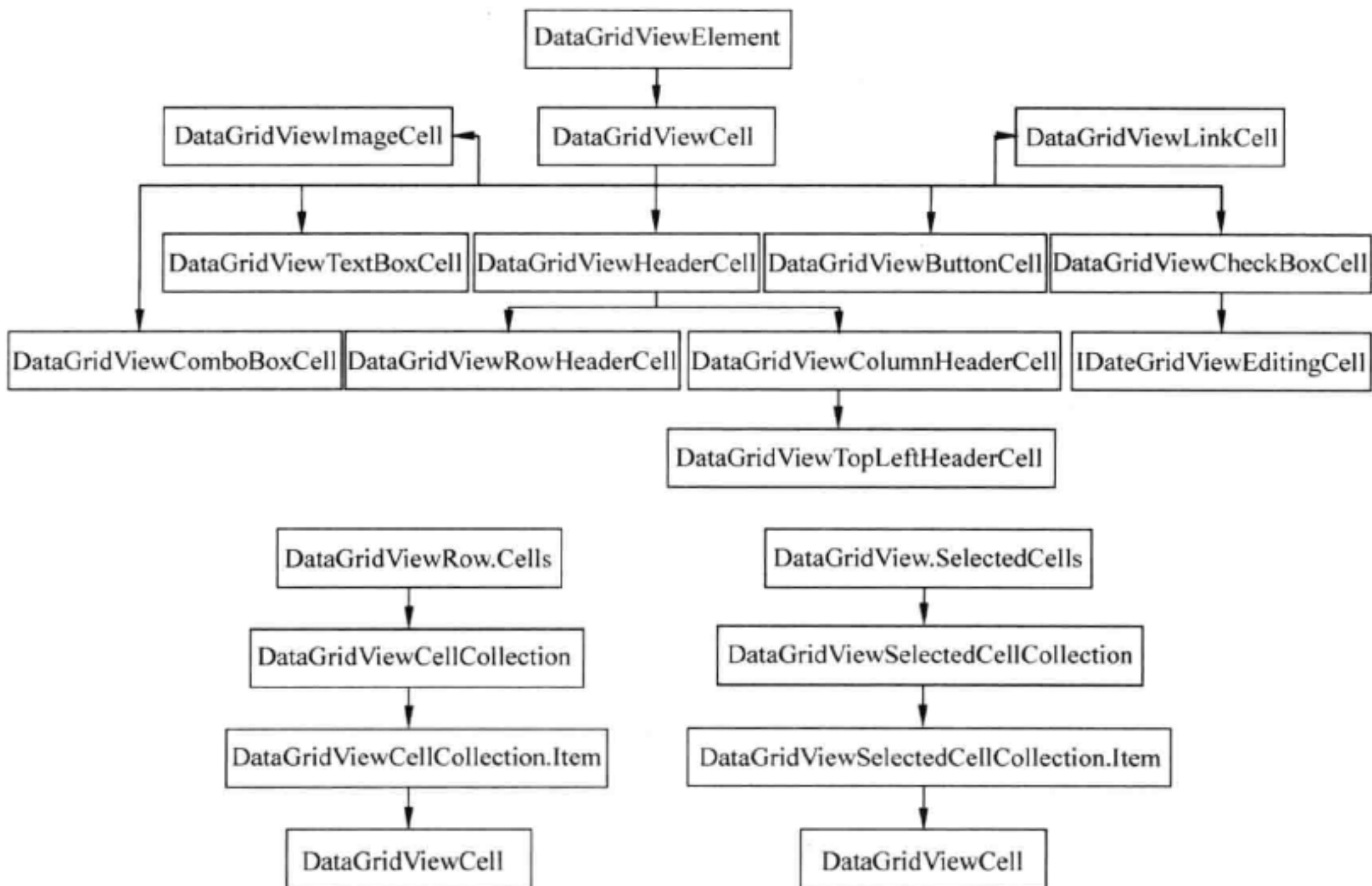


图 15.17 DataGridViewCell 继承层次结构

说明：DataGridView 控件通常通过单元格来输入数据。但是，DataGridViewCell 并不是 Windows 窗体的控件，它不能像 Windows 窗体控件那样可以在窗体设计器上对它的外观进行定义，它甚至没有控制自己的外观和绘制功能。

15.2.3 数据列对象模型 DataGridViewColumn

`DataGridViewColumn` 类就是在 `DataGridView` 中数据列的基础类型。在 `DataGridView` 控件中，既可以通过 `Columns` 属性访问 `DataGridView` 控件中的所有列，也可以使用 `SelectedColumns` 属性访问选定的列，如图 15.18 所示为 `DataGridViewColumn` 继承层次结构。对应图 15.17 可以看到，一些主要的单元格类型均具有相应的列类型。这些列类型都是从 `DataGridViewColumn` 这个基类派生出来的。从 `DataGridViewColumn` 派生出来的类有：`DataGridViewButtonColumn`、`DataGridViewCheckBoxColumn`、`DataGridViewComboBoxColumn`、`DataGridViewImageColumn`、`DataGridViewTextBoxColumn` 和 `DataGridViewLinkColumn`。

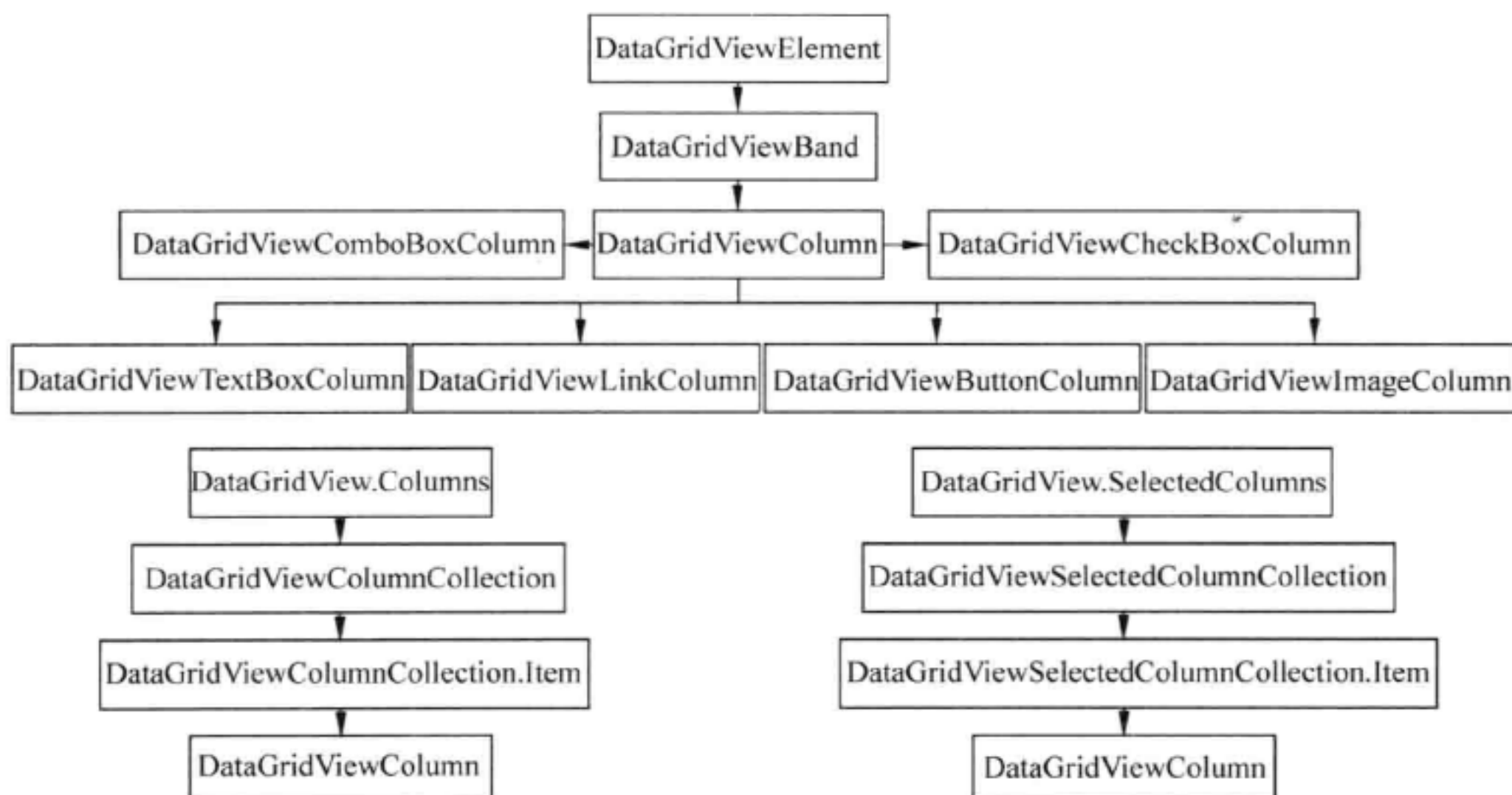


图 15.18 DataGridViewColumn 继承层次结构

15.2.4 数据行对象模型 DataGridViewRow

`DataGridViewRow` 类就是 `DataGridView` 中数据行的基础类型。在 `DataGridView` 控件中, 可以通过使用 `Rows` 属性访问 `DataGridView` 控件的行, 也可以使用 `SelectedRows` 属性访问选定的行。如图 15.19 所示为 `DataGridViewRow` 继承层次结构。

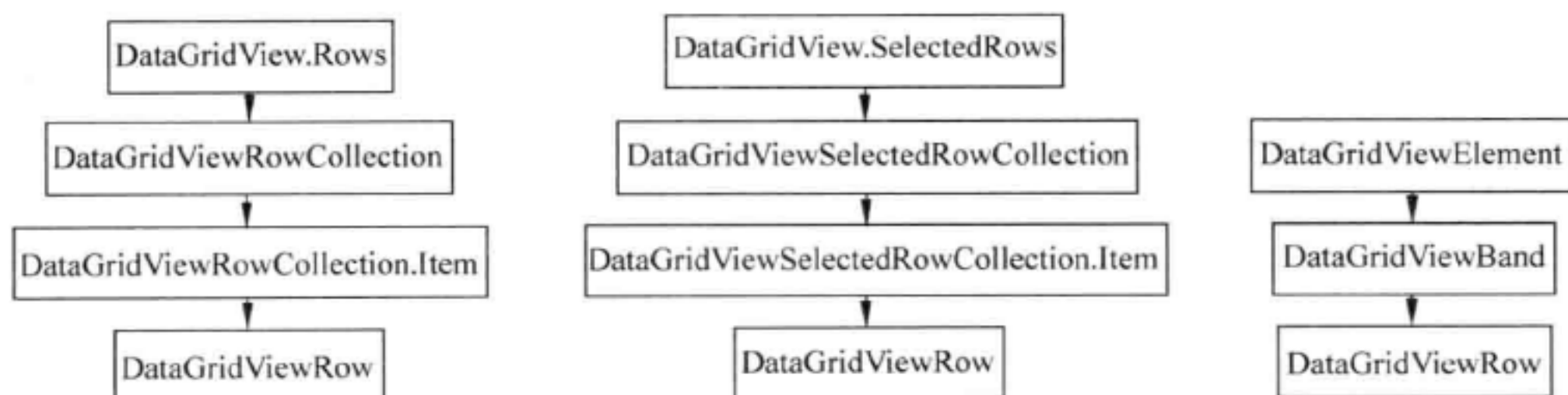


图 15.19 DataGridViewRow 继承层次结构


 **注意：**如果将 DataGridView 控件的 AllowUserToAddRows 属性设置为 true，则将会在最后一行显示一个用于添加新行的特殊行，此行也是 DataGridView 控件的 Rows 集合的一部分。

15.3 设置 DataGridView 的格式

DataGridView 的类中拥有大量的属性、方法和事件，通过这些 DataGridView 类可以提供更好的可配置性，开发人员可以通过对这些属性进行设置，从而定制出具有一定外观特性的 DataGridView 控件。也可以灵活地使用和重写 .NET 提供的各种方法和事件处理程序，配置出符合用户需求的 DataGridView 控件。下面的各小节中，将分别针对 DataGridView 控件的属性、方法和事件进行详细的说明。

15.3.1 设置单元格边框样式

DataGridView 控件的灵活性所表现出的一点是开发人员可以自定义 DataGridView 控件的边框和网格线的外观，并以此提高所开发产品的用户友好性。

 **说明：**在 DataGridView 控件边框设置中不但可以设置单元格的样式，还可以设置网格线的颜色，并可对不同的单元格应用不同的单元格边框样式，如可为左上角单元格、列标题单元格、行标题单元格和普通单元格分别设置不同的边框样式。

下面，将针对与 DataGridView 控件的单元格边框样式有关的属性进行阐述。

为了更好地了解 DataGridView 控件与单元格边框样式相关的属性，首先要了解两个与其相关的枚举类型。

- **BorderStyle** 是用来制定边框样式的枚举类型，通过使用此枚举的成员可以为单元格或其他具有边框的控件设置边框样式。其具体成员列表如表 15.1 所示。

表 15.1 BorderStyle 枚举成员列表

成 员	说 明	成 员	说 明
Fixed3D	三维边框	None	无边框
FixedSingle	单行边框		

- **DataGridViewAdvancedBorderStyle** 类可用于对 DataGridView 控件中的单元格的边框样式进行设置。通过设置 DataGridViewAdvancedBorderStyle 的 Left、Right、Top 和 Bottom 属性返回一个 DataGridViewAdvancedCellBorderStyle 枚举值类型对象，可分别设置单元格的左边框、右边框、上边框和下边框，也可以通过设置 DataGridViewAdvancedBorderStyle 的 All 属性来设置单元格的所有边框的边框样式。如果开发人员没有将 DataGridViewAdvancedBorderStyle 的 Left、Right、Top 和 Bottom 属性设置为相同值时，则 DataGridViewAdvancedBorderStyle 的 All 属性值只能为 NotSet（未设置边框）。
- **DataGridViewAdvancedCellBorderStyle** 是用来制定 DataGridView 单元格边框样式的枚举类型。它与 BorderStyle 枚举类型的区别是，DataGridViewAdvancedCellBorderStyle 只能用来设置 DataGridView 单元格边框样式，而 BorderStyle 可用来设置所有具有边框的控件。其具体成员列表如表 15.2 所示。

表 15.2 DataGridViewAdvancedCellStyle 成员列表

成 员	说 明	成 员	说 明
Insert	单边凹陷边框	OutSet	单线突起边框
InsertDouble	双线凹陷边框	OutSetDouble	双线突起边框
None	无边框	OutsetPartial	包含突起部分的单线边框
NotSet	未设置边框	Signle	单行边框

- DataGridViewCellStyle 是用来制定 DataGridView 的单元格边框样式的枚举类型。通过此枚举类型可设置 DataGridView 控件里的 CellBorderStyle 属性。DataGridViewCellStyle 枚举类型与 DataGridViewAdvancedCellStyle 枚举类型的区别在于，当使用 DataGridViewCellStyle 对 DataGridView 控件的单元格边框样式进行设置时，应根据此边框是行之间的边框还是列之间的边框而需要设置不同的成员值，不可通用。而使用 DataGridViewAdvancedCellStyle 对 DataGridView 控件的单元格边框样式进行设置时不需要考虑此问题。DataGridViewCellStyle 的具体成员列表如表 15.3 所示。

表 15.3 DataGridViewCellStyle 成员列表

成 员	说 明	成 员	说 明
Custom	已存在的自定义边框	SingleHorizontal	水平单行边框
None	无边框	SingleVertical	垂直单行边框
Raised	三维凸起边框	Sunken	三维凹陷边框
RaisedHorizontal	水平三维凸起边框	SunkenHorizontal	水平三维凹陷边框
RaisedVertical	垂直三维凸起边框	SunkenVertical	垂直三维凹陷边框
Single	单行边框		

- DataGridViewHeaderBorderStyle 是用来制定 DataGridView 的列标题单元格边框样式及行标题单元格边框样式的枚举类型。通过此枚举类型可设置 DataGridView 控件里的 ColumnHeadersBorderStyle 和 RowHeadersBorderStyle 属性。DataGridViewHeaderBorderStyle 的具体成员列表如表 15.4 所示。

表 15.4 DataGridViewHeaderBorderStyle 成员列表

成 员	说 明	成 员	说 明
Custom	已存在的自定义边框	Single	单行边框
None	无边框	Sunken	三维凹陷边框
Raised	三维凸起边框		


15.3.2 设置 DataGridView 控件边框样式

BorderStyle 属性返回的是一个 BorderStyle 枚举类型的数据，此属性主要是用来对 DataGridView 控件的边框样式进行获取和设置。对 DataGridView 控件的边框进行设置的函数代码如下所示。

```
private void setDataGridViewBorderStyle ( )
{
```



```
this.dataGridView1.BorderStyle = BorderStyle.Fixed3D;
}
```


说明: BorderStyle 属性主要是用来对整个 DataGridView 控件的边框进行设置, 如果需要对 DataGridView 控件的每个单元格的边框进行设置, 则需要使用 CellBorderStyle 属性。

15.3.3 设置单元格边框样式

CellBorderStyle 属性返回的是一个 DataGridViewCellBorderStyle 枚举类型的数据, 此属性主要是用来对 DataGridView 控件的单元格边框样式进行获取。对 DataGridView 控件的单元格边框样式进行设置的函数的代码如下所示。

```
private void setDataGridViewCellBorderStyle ( )
{
    this.dataGridView1.CellBorderStyle =DataGridViewCellBorderStyle.None;
}
```

RowHeadersBorderStyle 属性返回的是一个 DataGridViewHeaderBorderStyle 枚举类型的数据, 此属性主要是用来对 DataGridView 控件的行标题单元格边框样式进行获取或设置。

说明: 如果 RowHeadersBorderStyle 的属性设置的并不是 Single, 则 DataGridView 控件的 GridColor 属性必须设置为系统颜色。

RowHeadersBorderStyle 属性不能被设置为 Custom 值, 该值是一个只读值, 指示边框样式已通过使用 AdvancedRowHeadersBorderStyle 属性进行了自定义。对 DataGridView 控件的行标题单元格边框样式进行设置的函数的代码如下:

```
private void setDataGridViewRowHeadersBorderStyle ( )
{
    this.dataGridView1.RowHeadersBorderStyle=
        DataGridViewHeaderBorderStyle.Single;
}
```

ColumnHeadersBorderStyle 属性返回的是一个 DataGridViewHeaderBorderStyle 枚举类型的数据, 此属性主要是用来对 DataGridView 控件的列标题单元格边框样式进行获取或设置。另外, 如果 ColumnHeadersBorderStyle 的属性设置的值并不是 Single, 则 DataGridView 控件的 GridColor 属性必须设置为系统颜色。

ColumnHeadersBorderStyle 属性不能被设置为 Custom 值, 该值是一个只读值, 指示边框样式已通过使用 AdvancedColumnHeadersBorderStyle 属性进行了自定义。对 DataGridView 控件的列标题单元格边框样式进行设置的函数代码如下所示。

```
private void setDataGridViewColumnHeadersBorderStyle ( )
{
    this.dataGridView1. ColumnHeadersBorderStyle=DataGridViewHeader-
        BorderStyle.Single;
}
```

可以使用 CellBorderStyle 属性来更改 DataGridView 控件中单元格边框的样式。另外, 也可以使用 AdvancedCellBorderStyle 属性来获取 DataGridView 控件中所有单元格的边框样

式。AdvancedCellBorderStyle 属性返回的是一个 DataGridViewAdvancedBorderStyle 类型的数据。

如果开发人员希望在 DataGridView 控件中对单元格进行自定义的边框设置,以使该控件具有更加友好的外观,则可对 DataGridViewAdvancedBorderStyle 属性进行重写。重写的代码如下:

```
//对 DataGridViewAdvancedBorderStyle 属性进行重写
public override DataGridViewAdvancedBorderStyle AdjustedCellBorderStyle
{
    get
    {
        //对单元格的每个边框的外观进行设置
        DataGridViewAdvancedBorderStyle customStyle =
            new DataGridViewAdvancedBorderStyle();
        customStyle.Top = DataGridViewAdvancedCellBorderStyle.OutSet;
        customStyle.Left = DataGridViewAdvancedCellBorderStyle.OutSet;
        customStyle.Bottom = DataGridViewAdvancedCellBorderStyle.OutSet;
        customStyle.Right = DataGridViewAdvancedCellBorderStyle.OutSet;
        return customStyle;
    }
}
```


15.3.4 设置左上角单元格边框样式

AdjustedTopLeftHeaderBorderStyle 属性返回的是一个 DataGridViewAdvancedBorderStyle 类型的数据,它可以用来获取或设置 DataGridView 控件左上角单元格的边框样式。在 DataGridView 控件中,左上角单元格是一个较为特殊的单元格,它位于 DataGridView 控件中列标题所在行,以及行标所在列的交叉处。

如果开发人员希望在 DataGridView 控件中对左上角的单元格进行特殊的边框设置,以使 DataGridView 控件具有更加友好的外观,则可对 AdjustedTopLeftHeaderBorderStyle 属性进行重写。重写的代码如下:

```
public override DataGridViewAdvancedBorderStyle AdjustedTopLeftHeader-
BorderStyle
{
    get
    {
        DataGridViewAdvancedBorderStyle customStyle =
            new DataGridViewAdvancedBorderStyle();
        customStyle.Top = DataGridViewAdvancedCellBorderStyle.Insert;
        customStyle.Left = DataGridViewAdvancedCellBorderStyle.Insert;
        customStyle.Bottom = DataGridViewAdvancedCellBorderStyle.None;
        customStyle.Right = DataGridViewAdvancedCellBorderStyle.None;
        return customStyle;
    }
}
```

可以使用 ColumnHeadersBorderStyle 属性来更改 DataGridView 控件中列标题的单元格边框的样式。ColumnHeadersBorderStyle 属性返回的是一个 DataGridViewAdvancedBorderStyle 类型的数据。

 **技巧:** 可以使用 AdvancedColumnHeadersBorderStyle 属性来获取 DataGridView 控件中列标题的单元格边框样式。

如果开发人员希望在 DataGridView 控件中对列标题单元格进行自定义的边框设置, 以使该控件具有更加友好的外观, 则可对 DataGridViewAdvancedBorderStyle 属性进行重写。重写的代码如下:

```
public override DataGridViewAdvancedBorderStyle AdjustedColumnHeaders-
BorderStyle
{
    get
    {
        DataGridViewAdvancedBorderStyle customStyle =
            new DataGridViewAdvancedBorderStyle();
        customStyle.Top = DataGridViewAdvancedCellStyle.OutsetPartial;
        customStyle.Left = DataGridViewAdvancedCellStyle.OutsetPartial;
        customStyle.Bottom = DataGridViewAdvancedCellStyle.None;
        customStyle.Right =
            DataGridViewAdvancedCellStyle.OutsetPartial;
        return customStyle;
    }
}
```

AdvancedRowHeadersBorderStyle 属性返回的是一个 DataGridViewAdvancedBorderStyle 类型的数据, 它可以用来获取或设置 DataGridView 控件中 DataGridViewColumnHeaderCell 对象的边框样式。另外, 也可以使用 CellBorderStyle 属性来更改边框的样式。同样, 也可以使用 ColumnHeadersBorderStyle 来更改 DataGridViewColumnHeaderCell 对象的边框样式。

可以使用 RowHeadersBorderStyle 属性来更改 DataGridView 控件中行标题的单元格边框的样式。另外, 也可以使用 AdvancedRowHeadersBorderStyle 属性来获取 DataGridView 控件中行标题的单元格边框样式。ColumnHeadersBorderStyle 属性返回的是一个 DataGridViewAdvancedBorderStyle 类型的数据。

如果开发人员希望在 DataGridView 控件中对行、列标题单元格进行自定义的边框设置, 以使该控件具有更加友好的外观, 则可对 DataGridViewAdvancedBorderStyle 属性进行重写。重写的代码如下:

```
public override DataGridViewAdvancedBorderStyle AdjustedRowHeadersBorderStyle
{
    get
    {
        DataGridViewAdvancedBorderStyle customStyle =
            new DataGridViewAdvancedBorderStyle();
        customStyle.Top = DataGridViewAdvancedCellStyle.OutsetPartial;
        customStyle.Left = DataGridViewAdvancedCellStyle.OutsetPartial;
        customStyle.Bottom =
            DataGridViewAdvancedCellStyle.OutsetPartial;
        customStyle.Right = DataGridViewAdvancedCellStyle.None;
        return customStyle;
    }
}
```

15.3.5 设置单元格样式

在前面的属性介绍中可以看出, DataGridView 控件内的每个单元格都可以通过设置不同的属性拥有自己的框架样式。同样, DataGridView 控件内的每个单元格也可以通过设置


不同的属性拥有自己的样式，如背景色、前景色、文本格式和字体。

但是，如同 DataGridView 控件的单元格的框架样式可以集体设置一样（例如，可同时设置 DataGridView 控件的列标题单元格），DataGridView 控件内的多个单元格通常会共享特定的样式特征。

当某单元格位于某特定行或列内，或包含有特定的值时，这些具有某一固定特性的单元格就可成为可共享样式的单元格组。

有时，可共享样式的单元格存在重叠情况。这时，在发生重叠位置的单元格就要从多处获取其样式信息。有时这种特性可满足用户的特定需求，例如，如果希望 DataGridView 控件中的每个单元格都使用相同字体，只有货币列的单元格使用货币格式，并且只有带负数的货币单元格使用红色前景色。但有的时候，它会造成单元格样式的混乱，如何正确使用单元格继承样式及其样式继承逻辑，将在后面进行详细的说明。

为了研究 DataGridView 控件的单元格样式设置，首先需要了解的是 DataGridViewCellStyle 类。该类是用来表示应用到 DataGridView 控件中的各个单元格的样式信息和格式设置。

 **说明：**通过使用 DataGridViewCellStyle 类可以使开发者在多个 DataGridView 控件的单元格、行、列，以及行或列的标头之间共享样式信息。这样做，可以有效地避免为每个单元格分别设置样式属性所造成的内存消耗。


DataGridViewCellStyle 类中包含有许多与视觉样式相关的属性，下面将对它们进行说明。

BackColor 属性，对 DataGridView 控件中的单元格的背景色进行获取和设置。ForeColor 属性，对 DataGridView 控件中的单元格的前景色进行获取和设置。在设置时，ForeColor 的属性值最好与 BackColor 的属性值形成鲜明的对比。具体的设置方法如下面的代码所示。

```
private void setDataGridViewColor()
{
    //设置单元的背景色
    dataGridView1.DefaultCellStyle.BackColor = Color.White
    //设置单元的前景色
    dataGridView1.DefaultCellStyle.ForeColor= Color.Black
}
```

SelectionBackColor 属性，对 DataGridView 控件中的单元格在被选定时背景色进行获取和设置。在设置时，该属性值最好与 BackColor 的属性值形成鲜明的对比。

SelectionForeColor 属性，对 DataGridView 控件中的单元格在被选定时前景色进行获取和设置。

 **技巧：**在设置时，SelectionForeColor 的属性值最好与 SelectionBackColor 的属性值形成鲜明的对比。

具体的设置方法如下面的代码所示。

```
//设置单元格被选定时颜色
private void setDataGridViewSelectionColor()
{
    dataGridView1.DefaultCellStyle.SelectionBackColor= Color.Blue
}
```



```
dataGridView1.DefaultCellStyle.SelectionForeColor= Color.White
}
```

Font 属性, 对 DataGridView 控件中的单元格文本内容的字体进行获取和设置。在设置时, SelectionForeColor 的属性值最好与 SelectionBackColor 的属性值形成鲜明的对比。在开发过程中, 开发人员可以使用该属性更改应用于单元格文本的字体样式。因为 Font 属性为只读, 因此它是不可变的, 所以不能直接更改所返回对象的属性, 只能通过将新的或现有的 Font 赋值给该属性来修改它。具体的设置方法如下面的代码所示。

```
//设置单元格的字体
private void setDataGridViewFont()
{
    this.dataGridView1.DefaultCellStyle.Font = new Font("Tahoma", 12);
}
```

Alignment 属性, 对 DataGridView 控件中的单元格内容的对齐方式进行获取和设置。其值是 DataGridViewContentAlignment 的属性值之一。DataGridViewContentAlignment 是用来指定 DataGridView 单元格中内容的对齐方式的枚举类型。通过此枚举类型可设置 DataGridView 控件里的 Alignment 属性。DataGridViewContentAlignment 的具体成员列表如表 15.5 所示。

表 15.5 DataGridViewContentAlignment 成员列表

成 员	说 明
BottomCenter	DataGridView 单元格中的内容与单元格的中间水平对齐, 与单元格的底部垂直对齐
BottomLeft	DataGridView 单元格中的内容与单元格的左侧水平对齐, 与单元格的底部垂直对齐
BottomRight	DataGridView 单元格中的内容与单元格的右侧水平对齐, 与单元格的底部垂直对齐
TopCenter	DataGridView 单元格中的内容与单元格的中间水平对齐, 与单元格的顶部垂直对齐
TopLeft	DataGridView 单元格中的内容与单元格的左侧水平对齐, 与单元格的顶部垂直对齐
TopRight	DataGridView 单元格中的内容与单元格的右侧水平对齐, 与单元格的顶部垂直对齐
MiddleCenter	DataGridView 单元格中的内容与单元格的水平和垂直中心对齐
MiddleLeft	DataGridView 单元格中的内容与单元格的左侧水平对齐, 与单元格的中间垂直对齐
MiddleRight	DataGridView 单元格中的内容与单元格的右侧水平对齐, 与单元格的中间垂直对齐
NotSet	未设定对齐方式

对 Alignment 属性的具体设置方法如下所示。


```
//设置单元格中文字的布局
private void setDataGridViewAlignment()
{
    this.dataGridView1.DefaultCellStyle.Alignment=
        DataGridViewContentAlignment.MiddleCenter;
}
```

WrapMode 属性, 当 DataGridView 单元格中的文本内容太长而不能放在单行中时, WrapMode 属性指示将其换到下一行还是将其截断。如果 WrapMode 的属性值是 false, 则单元格将在单行上显示文本。如果 WrapMode 的属性值是 true, 则单元格将换行符显示为分行符, 并对超出单元格宽度的所有行进行换行。具体的设置方法如下面的代码所示。

```
private void setDataGridViewWrapMode()
{
```



```
//设置在 dataGridView1 中可以对文本超出部分进行换行
this.dataGridView1.DefaultCellStyle.WrapMode= true;
}
```

说明：可以通过使用 DataGridViewCellStyle 对象对 DataGridView、DataGridViewColumn、DataGridViewRow 和 DataGridViewCell 类及其派生类的相关属性进行设置。如果这些属性中有的值尚未设置，则开发人员可以实例化自己的 DataGridViewCellStyle 对象，并将这些对象分配给此属性。

15.3.6 单元格样式设置的优先级

下面将对 DataGridView、DataGridViewColumn、DataGridViewRow 和 DataGridViewCell 类及其派生类需要用 DataGridViewCellStyle 对象进行设置的相关属性进行逐一说明。

DefaultCellStyle 属性，在未设置其他的单元格样式属性的情况下，该属性用于获取或设置应用于 DataGridView 中所有单元格的默认单元格样式。具体的设置方法如下：

```
private void setDataGridViewDefaultCellStyle ()
{
    //设置 DataGridViewCellStyle 的属性
    this.dataGridView1.DefaultCellStyle.BackColor = Color.Gray;
    this.dataGridView1.DefaultCellStyle.Font = new Font("Arial", 10);
    //创建和初始化新的 DataGridViewCellStyle 对象以供多个行和列使用
    DataGridViewCellStyle highlightCellStyle = new DataGridViewCellStyle();
    highlightCellStyle.BackColor = Color.Blue;
    highlightCellStyle.ForeColor = Color.White;
    DataGridViewCellStyle currencyCellStyle = new DataGridViewCellStyle();
    currencyCellStyle.Format = "C";
    currencyCellStyle.BackColor = Color.Red;
    currencyCellStyle.ForeColor = Color.Green;
    //设置特定行和列的 DefaultCellStyle 属性
    this.dataGridView1.Rows[3].DefaultCellStyle = highlightCellStyle;
    this.dataGridView1.Rows[8].DefaultCellStyle = highlightCellStyle;
    this.dataGridView1.Columns[0].DefaultCellStyle =currencyCellStyle;
    this.dataGridView1.Columns[1].DefaultCellStyle =currencyCellStyle;
}
```

DataGridView 控件中，通过 DefaultCellStyle 属性指定的样式会影响所有的单元格，但是有的单元格被其他属性所指定的样式重写，此时 DefaultCellStyle 属性对这些单元格的样式不加影响。这些属性分别为：DataGridViewColumn.DefaultCellStyle、RowsDefaultCellStyle、AlternatingRowsDefaultCellStyle、DataGridViewRow.DefaultCellStyle 和 DataGridViewCell.Style。

ColumnHeadersDefaultCellStyle 属性用于获取或设置控件的列标头使用的默认单元格样式；RowHeadersDefaultCellStyle 属性用于获取或设置控件的行标头使用的默认单元格样式。具体的设置方法如下：

```
private void setDataGridViewHeadersDefaultCellStyle()
{
    //创建和初始化新的 DataGridViewCellStyle 对象以供列标头和行标头使用
    DataGridViewCellStyle columnHeaderCellStyle = new DataGridViewCellStyle();
    columnHeaderCellStyle.BackColor = Color.Blue;
    columnHeaderCellStyle.ForeColor = Color.White;
    DataGridViewCellStyle rowHeaderCellStyle = new DataGridViewCellStyle();
}
```



```

rowHeaderCellStyle.BackColor = Color.Red;
rowHeaderCellStyle.ForeColor = Color.Green;
//设置列标头和行标头
this.dataGridView1.ColumnHeadersDefaultCellStyle =
    columnHeaderCellStyle;
this.dataGridView1.RowHeadersDefaultCellStyle = rowHeaderCellStyle;
}

```

RowsDefaultCellStyle 属性用于获取或设置控件中所有行使用的默认单元格样式，但是并不包括标头单元格。AlternatingRowsDefaultCellStyle 属性用于获取或设置应用于 DataGridView 控件的奇数行的默认单元格样式。具体的设置方法如下：

```

private void setDataGridViewHeadersDefaultCellStyle()
{
    //创建和初始化新的 DataGridViewCellStyle 对象以供行和奇数行使用
    DataGridViewCellStyle rowsDefaultCellStyle = new DataGridViewCellStyle();
    rowsDefaultCellStyle.BackColor = Color.Blue;
    rowsDefaultCellStyle.ForeColor = Color.White;
    DataGridViewCellStyle alternatingRowsDefaultCellStyle =
        new DataGridViewCellStyle();
    alternatingRowsDefaultCellStyle.BackColor = Color.Red;
    alternatingRowsDefaultCellStyle.ForeColor = Color.Green;
    //设置行样式和奇数行样式
    this.dataGridView1.RowsDefaultCellStyle= rowsDefaultCellStyle;
    this.dataGridView1.AlternatingRowsDefaultCellStyle=
        alternatingRowsDefaultCellStyle;
}

```


在应用系统中，为了能够让用户清楚地分辨出表格中每一行中的单元格，经常对表格进行交替行的不同颜色的背景色设置。对于有多列的宽表，这一功能非常实用。使用 DataGridView 控件，可以指定交替行的完整样式信息，使用户能够使用背景色、前景色和字体等样式特征来区分交替行。下面的代码通过设置 DataGridView 的 RowsDefaultCellStyle 属性和 AlternatingRowsDefaultCellStyle 属性来实现上面的功能。

```

this.dataGridView1.RowsDefaultCellStyle.BackColor = Color.Red;
this.dataGridView1.AlternatingRowsDefaultCellStyle.BackColor =Color.Green;

```

在 DataGridView 控件中，如果以前没有设置过任何样式属性，那么在绘制过程中，当需要获取样式属性值时，将自动生成一个新的 DataGridViewCellStyle 对象的实例。

说明：在多个 DataGridView 元素间共享 DataGridViewCellStyle 对象自然可以避免不必要地重复样式信息，但是，当不同的样式设置间存在共享区时（例如，使一个单元格同时具有 ColumnHeadersDefaultCellStyle 和 DefaultCellStyle 两种样式时，该如何显示），DataGridView 控件将从控件级向下筛选至列级，再到行级，然后到单元格级。也就是说，如果在单元格级未设置某一特定 DataGridViewCellStyle 属性，则使用行级的默认属性设置。如果在行级也未设置该属性，则使用默认列级设置。最后，如果在列级也未设置该属性，则使用默认的 DataGridView 设置。使用此设置可避免必须在多个级别重复属性设置。在每一个级别，只需简单地指定不同于上面各级的样式。


15.3.7 设置 DataGridView 外观

DataGridView 控件的 ColumnHeadersHeight 属性可以用来设置列标题行的高度，以像素为单位。DataGridView 控件的 ColumnHeadersHeightSizeMode 属性用来指示是否可以调整列标题的高度。该属性还可指示出 DataGridView 控件是由用户调整高度还是根据标题的内容自动调整高度。当该属性为 AutoSize 时，用户不能自己调整列标题的高度。

ColumnHeadersVisible 属性指示了 DataGridView 控件是否能显示列标题行。在应用程序中，有时可能希望在显示 DataGridView 时并不显示相应的列标题。隐藏列标题只需将该属性设置为 false，具体如下面的代码所示。

```
dataGridView1.ColumnHeadersVisible = false;
```

GridColor 属性用于获取和设置 DataGridView 控件中的网格线的颜色，网格线是在 DataGridView 控件中对单元格进行分隔的线。

说明：在边框样式属性为 single 时，可以将 GridColor 属性设置为任何颜色；但是对于其他类型的边框，GridColor 属性将不再起作用，颜色由操作系统指定。

RowHeadersVisible 属性、RowHeadersWidth 属性和 RowHeadersWidthSizeMode 属性的使用与 ColumnHeadersVisible 属性、ColumnHeadersHeight 属性和 ColumnHeadersHeightSizeMode 属性几乎完全相同，在此不再赘述。

15.3.8 用属性设置可对表格的操作

AllowUserToAddRows 属性指示用户是否可以用 DataGridView 添加行。如果其值为 true，则 DataGridView 控件下面会自动出现一行新的没有任何数据信息的数据行，用户可以通过此行向 DataGridView 添加新的数据。

AllowUserToDeleteRows 属性是用来指示是否允许用户从 DataGridView 中删除行。当其值为 true 时，用户可以从 DataGridView 中删除行，当其值为 false 时，则用户不可以从 DataGridView 中删除行。

下面的例子将通过对 DataGridView 控件的 AllowUserToAddRows 的属性和 AllowUserToDeleteRows 属性进行设置，以使该控件所在的应用程序的用户不能向该控件添加和删除数据行。

```
private void setDataGridView ( )
{
    //用户不能向 DataGridView 控件添加和删除数据行
    this.dataGridView1.AllowUserToAddRows=false;
    this.dataGridView1.AllowUserToDeleteRows=false;
}
```

AllowUserToOrderColumns 属性是用来指示是否允许用户通过手动对列重新排序。当其值为 true 时，用户可以更改列的顺序，当其值为 false 时，则用户不可以更改列的顺序。

AllowUserToResizeColumns 属性是用来指示是否允许用户调整列的宽度。当其值为

true 时, 用户可以调整列的宽度, 当其值为 false 时, 则用户不可以调整列的宽度。

AllowUserToResizeRows 属性是用来指示是否允许用户调整行的高度。当其值为 true 时, 用户可以调整行的高度, 当其值为 false 时, 则用户不可以调整行的高度。

下面的例子将通过 DataGridview 控件的 AllowUserToOrderColumns 的属性、AllowUserToResizeRows 属性和 AllowUserToResizeColumns 的属性进行设置, 以使 DataGridview 控件所在的应用程序的用户可以对该控件中的列和行自动调整大小, 并能对列重新排序。

```
private void setDataGridView ( )
{
    //用户可以对 DataGridview 控件中的列和行自动调整大小, 并能够对列进行重新排序
    this.dataGridView1.AllowUserToOrderColumns=false;
    this.dataGridView1.AllowUserToResizeColumns=false;
    this.dataGridView1.AllowUserToResizeRows=false;
}
```

SortedColumn 属性指示了 DataGridview 控件中当前排序所依据的列。SortOrder 属性返回一个 SortOrder 枚举值, 用以指示以何种排序方式对 DataGridview 控件中的项进行排序。SortOrder 枚举类型的成员如表 15.6 所示。

表 15.6 SortOrder 成员列表

成 员	说 明	成 员	说 明
Ascending	按递增顺序排序	None	不排序
Descending	按递减顺序排序		

15.3.9 DataGridView 其他常用属性

Columns 属性返回一个 DataGridViewColumnCollection 类型的数据, 里面包含了 DataGridView 控件中所有列的集合。可以通过对此属性进行操作, 从而间接实现对 DataGridView 控件中列集合的操作, 具体使用方法如下:


```
private void setDataGridColumns ( )
{
    //获得 DataGridView1 中的第一列
    DataGridViewColumn column = this.dataGridView1.Columns.GetFirstColumn();
    //将此列从 dataGridView1 中移除
    this.dataGridView1.Columns.Remove(column);
    //判断此控件中是否还有 column 列
    if(this.dataGridView1.Columns.Contains(column))
    {
        //此语句段应该不会运行到
        //如果控件中还存在 column 列
        //获得此 column 列的列索引号
        int Index =.dataGridView1.Columns.IndexOf (column)
        //使用 RemoveAt () 方法将 column 列移除
        this.dataGridView1.Columns.RemoveAt (Index);
    }
    else
    {
        //如果控件中不存在 column 列
        //则在此控件中再添加此列
        this.dataGridView1.Columns.Add(column);
    }
}
```

```

    }
    //清空控件中的所有列
    this.dataGridView1.Columns.Clear();
}

```

ColumnCount 属性是一个整数值，返回的是 DataGridView 中显示的列数。其值比 DataGridView 控件中最后一列的索引值大 1。

 说明：this.dataGridView1.ColumnCount == this.dataGridView1.Columns.Count 该等式恒成立。


Rows 属性返回一个 DataGridViewRowCollection 类型的数据，里面包含了 DataGridView 控件中所有行的集合。可以通过对此属性进行操作，从而间接实现对 DataGridView 控件中行集合的操作，具体使用方法如下：

```

private void setDataGridRows ( )
{
    //获得 dataGridView1 中的第一行
    DataGridViewRow row = this.dataGridView1.Rows.GetFirstRow();
    //将此行从 dataGridView1 中移除
    this.dataGridView1.Rows.Remove(row);
    //判断此控件中是否还有 row 行
    if(this.dataGridView1.Rows.Contains(row))
    {
        //此语句段应该不会运行到
        //如果控件中还存在 row 行
        //获得此 row 行的行索引号
        int Index = .dataGridView1.Rows.IndexOf(row)
        //使用 RemoveAt() 方法将 column 列移除
        this.dataGridView1.Rows.RemoveAt(Index);
    }
    else
    {
        //如果控件中不存在 row 行
        //则在此控件中再添加此行
        this.dataGridView1.Rows.Add(row);
    }
    //清空控件中的所有行
    this.dataGridView1.Rows.Clear();
}

```

RowCount 属性是一个整数值，返回的是 DataGridView 中显示的行数。其值比 DataGridView 控件中最后一行的索引值大 1。

 说明：this.dataGridView1.RowCount == this.dataGridView1.Rows.Count 该等式恒成立。

FirstDisplayedCell 返回一个 DataGridViewCell 类型的对象，此单元格对象位于 DataGridView 中的第 1 行第 1 列中，即 DataGridView 控件左上角的单元格。CurrentCell 属性返回一个 DataGridViewCell 类型的对象，此对象是该控件中当前活动的单元格。默认值是该控件的 FirstDisplayedCell。在下面的代码中，就是将该控件的 FirstDisplayedCell 设置为该控件的 CurrentCell。

```

private void setCurrentCell()
{

```



```
this.dataGridView1.CurrentCell = this.dataGridView1.FirstDisplayedCell;
}
```

CurrentCellAddress 属性将返回一个 Point 类型的对象，用于标注此单元格在 DataGridView 控件中的位置。Point 类型是两个整数所组成的有序对，在这里是当前活动单元格的行索引和列索引。

CurrentRow 属性将返回一个 DataGridViewRow 类型的对象，此对象包含着 CurrentCell 的 DataGridViewRow。

NewRowIndex 属性返回的是一个 int 类型的数值，它标志着在 DataGridView 控件中可以由用户添加新的行，并获得所添加的新数据行的索引值。可以通过下面的代码对该属性进行获取：

```
private int getNewRowIndex()
{
    int newIndex = 0;
    //如果用户可以向 dataGridView1 添加行
    if( this.dataGridView1.AllowUserToAddRows)
    {
        //获得新添加行的索引号
        newIndex = this.dataGridView1.NewRowIndex;
    }
    return newIndex;
}
```

ShowCellToolTips 属性是一个 boolean 类型的值，它用来指示当鼠标指针停留在单元格上时，是否显示工具提示，具体的提示内容可以在 DataGridView 里的 ToolTip 属性中进行设置。对 ShowCellToolTips 的属性进行设置的代码如下所示。

```
private void setShowCellToolTips()
{
    //在 dataGridView1 中显示工具提示
    this.dataGridView1.ShowCellToolTips=true;
}
```

15.3.10 用 AutoResizeColumn 方法调列宽

AutoResizeColumn()方法主要是用来调整某一列的宽度。AutoResizeColumn()方法使用了重载机制来完成，一共有 3 种使用途径，分别介绍如下。

(1) 调整指定列的列宽，使列宽设置为可以使列中各单元格的内容均能显示，代码设置如下：

```
DataGridView.AutoResizeColumn (int)
```

□ 参数是指定的调节大小的列的索引号，通常从 0 开始，最大的索引号比列数小 1。

(2) 按照指定的调整方法调整指定列的列宽，代码设置如下：

```
DataGridView.AutoResizeColumn (int, DataGridViewAutoSizeColumnMode)
```

□ int 型参数是指定调节大小的列的索引号，通常从 0 开始，最大是索引号比列数小 1。

□ DataGridViewAutoSizeColumnMode 型参数是一个 DataGridViewAutoSizeColumn-

Mode 类型的枚举类型，用于指定 DataGridView 控件如何调整指定列的列宽。

(3) 按照指定的调整方法调整指定列的列宽，并可以由开发人员确定，列中各行的行高是否可调，代码设置如下：


```
DataGridView.AutoSizeColumn (int, DataGridViewAutoSizeColumnMode, bool)
```

- ❑ int 型参数是指定的调节大小的列的索引号，通常从 0 开始，最大的索引号比列数小 1。
- ❑ DataGridViewAutoSizeColumnMode 型参数是一个 DataGridViewAutoSizeColumnMode 类型的枚举类型，用于指定 DataGridView 控件如何调整指定列的列宽。
- ❑ boolean 类型的参数用于指定是否可以对列中各行的高度进行调整。如果列中各行的高度不可以调整，也就是说要基于现有行高进行调整，则 boolean 类型的变量为 true；如果列中各行的高度可以调整，也就是说不基于现有行高进行调整，则 boolean 类型的变量为 false。

DataGridViewAutoSizeColumnMode 枚举值，用以指示以何种方式对 DataGridView 控件中指定列的列宽进行调整。DataGridViewAutoSizeColumnMode 枚举类型的成员如表 15.7 所示。

表 15.7 DataGridViewAutoSizeColumnMode 成员列表

成 员	说 明
AllCells	使列宽设置为可以使列中各单元格的内容均能显示
AllCellsExceptHeader	使列宽设置为可以使列中除标题单元格外的所有单元格的内容均能显示
ColumnHeader	使列宽设置为可以使列中标题单元格的内容正常显示
DisplayedCells	使列宽设置为可以使列中目前显示在显示屏上的各单元格的内容均能显示
DisplayedCellsExceptHeader	使列宽设置为可以使列中目前显示在显示屏上的，除标题单元格外的所有单元格的内容均能显示
Fill	使所有的列宽度和正好与 DataGridView 控件显示单元格区域的宽度相同
None	列宽不发生调整
NotSet	未设置

 说明：AutoSizeRow() 方法主要是用来调整某一行的高度。

AutoSizeRow() 方法的使用方法和 AutoResizeColumn() 方法类似，不同之处就是将 DataGridViewAutoSizeColumnMode 型参数变为了 DataGridViewAutoSizeRowMode 型参数。

DataGridViewAutoSizeRowMode 型参数是一个 DataGridViewAutoSizeRowMode 枚举类型的对象。用以指示以何种方式对 DataGridView 控件中指定行的高度进行调整。DataGridViewAutoSizeRowMode 枚举类型的成员如表 15.8 所示。

表 15.8 DataGridViewAutoSizeRowMode 成员列表

成 员	说 明
AllCells	使行高设置为可以使行中各单元格的内容均能显示
AllCellsExceptHeader	使行高设置为可以使行中除标题单元格外的所有单元格的内容均能显示
RowHeader	使行高设置为可以使行中标题单元格的内容正常显示

15.3.11 用 AutoResizeColumns 方法调整所有列宽

AutoResizeColumns()方法主要是用来调整所有列的宽度。AutoResizeColumns()方法使用了重载机制来完成，一共有 3 种使用途径，分别介绍如下。

(1) 调整所有列的列宽，使列宽设置的可以使所有列中的各单元格的内容均能显示，代码设置如下：

```
DataGridView. AutoResizeColumns()
```

(2) 按照指定的调整方法调整指定列的列宽，代码设置如下。

```
DataGridView. AutoResizeColumns(DataGridViewAutoSizeColumnsMode)
```

❑ DataGridViewAutoSizeColumnsMode 型参数是一个 DataGridViewAutoSizeColumnsMode 类型的枚举类型，用于指定 DataGridView 控件如何调整所有列的列宽。

(3) 按照指定的调整方法调整所有列的列宽，并可以由开发人员确定，各行的行高是否可调，代码设置如下：

```
DataGridView. AutoResizeColumns(DataGridViewAutoSizeColumnsMode, bool)
```

❑ DataGridViewAutoSizeColumnsMode 型参数是一个 DataGridViewAutoSizeColumnsMode 类型的枚举类型，用于指定 DataGridView 控件如何调整所有列的列宽。

❑ boolean 类型的参数用于指定是否可以对各行的高度进行调整。如果各行的高度不可以调整，也就是说要基于现有行高进行调整，则 boolean 类型的变量为 true；如果各行的高度可以调整，也就是说不基于现有行高进行调整，则 boolean 类型的变量为 false。

DataGridViewAutoSizeColumnsMode 枚举值，用以指示以何种方式对 DataGridView 控件中指定列的列宽进行调整。DataGridViewAutoSizeColumnsMode 枚举类型的成员如表 15.9 所示。

表 15.9 DataGridViewAutoSizeColumnsMode 成员列表


成 员	说 明
AllCells	使列宽设置为可以使列中各单元格的内容均能显示
AllCellsExceptHeader	使列宽设置为可以使列中除标题单元格外的所有单元格的内容均能显示
ColumnHeader	使列宽设置为可以使列中标题单元格的内容正常显示
DisplayedCells	使列宽设置为可以使列中目前显示在显示屏上的各单元格的内容均能显示
DisplayedCellsExceptHeader	使列宽设置为可以使列中目前显示在显示屏上的除标题单元格外的所有单元格的内容均能显示
Fill	使所有的列宽度和正好与 DataGridView 控件的显示单元格的区域的宽度相同
None	列宽不发生调整

AutoResizeRows()方法的使用方法和 AutoResizeColumns()方法类似，不同之处就是将 DataGridViewAutoSizeColumnsMode 型参数变为了 DataGridViewAutoSizeRowsMode 型参

数。DataGridViewAutoSizeRowsMode 型参数是一个 DataGridViewAutoSizeRowsMode 枚举类型的对象。用以指示以何种方式对 DataGridView 控件中指定行的高度进行调整。DataGridViewAutoSizeRowsMode 枚举类型的成员如表 15.10 所示。

表 15.10 DataGridViewAutoSizeRowsMode 成员列表

成 员	说 明
AllCells	使行高设置为可以使行中各单元格的内容均能显示
AllCellsExceptHeader	使行高设置为可以使行中除标题单元格外的所有单元格的内容均能显示
AllHeader	使行高设置为可以使行中标题单元格的内容正常显示
DisplayedCells	使行高设置为可以使行中目前显示在显示屏上的各单元格的内容均能显示
DisplayedCellsExceptHeader	使行高设置为可以使行中目前显示在显示屏上的, 除标题单元格外的所有单元格的内容均能显示
DisplayedlHeaders	使行高设置的可以使行中目前显示在显示屏上的标题单元格的内容正常显示
None	行高不发生调整

 说明: AutoResizeRows() 方法主要是用来调整所有行的高度。

15.3.12 用 AutoResizeColumnHeadersHeight 方法调整列标题高度

AutoResizeColumnHeadersHeight() 方法主要是用来调整列标题的高度, 使列标题的内容可以正常显示。AutoResizeColumnHeadersHeight() 方法使用了重载机制来完成, 一共有 4 种使用途径, 分别介绍如下。

(1) 调整列标题的高度, 使列标题的内容可以正常显示, 代码设置如下:

```
DataGridView. AutoResizeColumnHeadersHeight()
```

(2) 调整列标题的高度, 使指定列的列标题的内容可以正常显示, 代码设置如下:

```
DataGridView. AutoResizeColumnHeadersHeight(int)
```

□ 参数是指定的列的索引号, 通常从 0 开始, 最大的索引号比列数小 1。

(3) 调整列标题的高度, 并可以由开发人员确定行标题的宽度是否可调, 列宽是否可调, 代码设置如下:

```
DataGridView. AutoResizeColumnHeadersHeight(bool, bool)
```

□ 第 1 个 boolean 类型的参数用于指示行标题的宽度是否可调, 如果可调, 则为 false; 如果不可调, 则为 true。此处与人类自然语言相违背, 请读者注意。

□ 第 2 个 boolean 类型的参数用于指示列宽是否可调, 如果可调, 则为 false; 如果不可调, 则为 true。此处与人类自然语言相违背, 请读者注意。

(4) 调整列标题的高度, 使指定列的列标题的内容可以正常显示, 并可以由开发人员确定各行的行高是否可调, 代码设置如下:

```
DataGridView. AutoResizeColumnHeadersHeight(int, bool, bool)
```

□ int 型参数是指定的列的索引号, 通常从 0 开始, 最大的索引号比列数小 1。

- ❑ 第 1 个 `boolean` 类型的参数用于指示行标题的宽度是否可调, 如果可调, 则为 `false`; 如果不可调, 则为 `true`。此处与人类自然语言相违背, 请读者注意。
- ❑ 第 2 个 `boolean` 类型的参数用于指示列宽是否可调, 如果可调, 则为 `false`; 如果不可调, 则为 `true`。此处与人类自然语言相违背, 请读者注意。

15.3.13 用 `AutoSizeRowHeadersWidth` 方法调整行标题宽度

`AutoSizeRowHeadersWidth()`方法主要是通过指定的调整方法来调整行标题的宽度。`AutoSizeRowHeadersWidth()`方法使用了重载机制来完成, 一共有 4 种使用途径, 分别介绍如下。

- (1) 按照指定的调整方法来调整行标题的宽度, 代码设置如下:

```
DataGridView.AutoSizeRowHeadersWidth(DataGridViewRowHeadersWidthSizeMode)
```

- ❑ `DataGridViewRowHeadersWidthSizeMode` 参数是一个 `DataGridViewRowHeadersWidthSizeMode` 类型的枚举类型, 用于指定 `DataGridView` 控件如何调整行标题的宽度。

- (2) 按照指定的调整方法来调整行标题的宽度, 使指定行的行标题内容可以完全显示, 代码设置如下:

```
DataGridView.AutoSizeRowHeadersWidth(int, DataGridViewRowHeadersWidthSizeMode)
```

- ❑ `int` 型参数是指定的行的索引号, 通常从 0 开始, 最大的索引号比行数小 1。
- ❑ `DataGridViewRowHeadersWidthSizeMode` 参数是一个 `DataGridViewRowHeadersWidthSizeMode` 类型的枚举类型, 用于指定 `DataGridView` 控件如何调整行标题的宽度。

- (3) 按照指定的调整方法来调整行标题的宽度, 并可以由开发人员确定列标题的宽度是否可调, 行宽是否可调, 代码设置如下:

```
DataGridView.AutoSizeRowHeadersWidth(DataGridViewRowHeadersWidthSizeMode, bool, bool)
```

- ❑ `DataGridViewRowHeadersWidthSizeMode` 参数是一个 `DataGridViewRowHeadersWidthSizeMode` 类型的枚举类型, 用于指定 `DataGridView` 控件如何调整行标题的宽度。
- ❑ 第 1 个 `boolean` 类型的参数用于指示列标题的宽度是否可调, 如果可调, 则为 `false`; 如果不可调, 则为 `true`。此处与人的自然语言相违背, 请读者注意。
- ❑ 第 2 个 `boolean` 类型的参数用于指示行宽是否可调, 如果可调, 则为 `false`; 如果不可调, 则为 `true`。此处与人类自然语言相违背, 请读者注意。

- (4) 按照指定的调整方法来调整行标题的宽度, 使指定行的行标题内容可以完全显示, 并可以由开发人员确定, 列标题的宽度是否可调, 行宽是否可调, 代码设置如下:


```
DataGridView.AutoSizeRowHeadersWidth(int, DataGridViewRowHeadersWidthSizeMode, bool, bool)
```

- ❑ `int` 型参数是指定的行的索引号，通常从 0 开始，最大的索引号比行数小 1。
- ❑ 第 1 个 `boolean` 类型的参数用于指示行标题的宽度是否可调，如果可调，则为 `false`；如果不可调，则为 `true`。此处与人类自然语言相违背，请读者注意。
- ❑ 第 2 个 `boolean` 类型的参数用于指示列宽是否可调，如果可调，则为 `false`；如果不可调，则为 `true`。此处与人类自然语言相违背，请读者注意。

`DataGridViewRowHeadersWidthSizeMode` 枚举类型的成员如表 15.11 所示。

表 15.11 `DataGridViewRowHeadersWidthSizeMode` 成员列表

成 员	说 明
<code>AutoSizeToAllHeaders</code>	使行宽设置为可以使各行的行标头内的内容均能显示
<code>AutoSizeToDisplayedHeaders</code>	使行宽设置为可以使行中目前显示在显示屏上的各行的行标头内的内容均能显示
<code>AutoSizeToFirstHeader</code>	使行宽设置为可以使第一行的行标头内的内容能完整显示
<code>DisableResizing</code>	用户不能调整列宽
<code>EnableResizing</code>	用户可以调整列宽

 说明：在前面提到的可以调整大小的方法，会在开发人员使用的时候按照其指定的方案进行大小的调节，每次调用，只调节一次相关的大小。如果读者想实现实时调节，即在单元格内容发生改变时进行调节，则可以通过设置相关的属性（如 `AutoSizeMode` 属性、`AutoSizeColumnsMode` 属性）来实现。

15.3.14 `DataGridView` 其他常用方法

`ClearSelection()` 方法主要是用来取消对当前某一个单元格的选定。`AutoResizeColumn()` 方法使用了重载机制来完成，一共有两种使用途径，分别介绍如下

- (1) 取消所有被选择的单元格，代码设置如下：

```
DataGridView.ClearSelection()
```

- (2) 取消所有未被特殊指出的被选择的单元格，代码设置如下：

```
DataGridView.ClearSelection(int, int, bool)
```

- ❑ `int` 型参数是指定不包含在内的单元格的列索引。
- ❑ `int` 型参数是指定不包含在内的单元格的行索引。
- ❑ 一个 `boolean` 类型的值，当它为 `true` 时，则将有前两个参数确定的单元格置为选定状态；当它为 `false` 时，则保持由前两个参数确定的单元的状态不变。

`DisplayedColumnCount()` 方法返回一个整数值，它表示在当前界面中显示出来的数据列数，具体的方法定义如下所示。


```
public int DisplayedColumnCount(bool)
```

- ❑ `boolean` 型参数是指定返回的列数值里是否应该包含未显示完全的列。

`DisplayedRowCount()` 方法返回一个整数值，它表示在当前界面中显示出的数据行数，具体的方法定义如下所示。

```
public int DisplayedRowCount(bool)
```


□ `boolean` 型参数是指定返回的行数值里是否应该包含未显示完全的行。

 **说明：**使用 `Sort()` 方法可以对 `DataGridView` 控件的内容进行排序，通常情况下，对 `DataGridView` 控件中的数据进行排序时需要指定排序所依照的数据列。

15.4 DataGridView 中显示数据的相关类

15.4.1 设置数据排序模式

在 `DataGridView` 的数据中，数据行的排列顺序通常是以 `DataGridView` 控件中的某一列的数据为基准对数据进行排序的。而 `DataGridViewColumn` 的 `SortMode` 属性则指定了数据是根据何种模式进行排序的。`SortMode` 属性值是一个 `DataGridViewColumnSortMode` 枚举类型值，该值的主要成员列表如表 15.12 所示。

表 15.12 `DataGridViewColumnSortMode` 成员列表

成 员	说 明
<code>Automatic</code>	单击列标头时会自动对该列的数据进行排序
<code>NotSortable</code>	不适合进行排序
<code>Programmatic</code>	以编程方式对该列进行排序


15.4.2 用 `DataGridViewTextBoxColumn` 显示文本数据列

`DataGridViewTextBoxColumn` 类似于 `TextBox` 控件，主要用于显示数字、字符串等文本型数据。当 `DataGridView` 控件是可编辑时，双击想要编辑的 `DataGridViewCell`，则相应的单元格就会被激活，用户可以像使用 `TextBox` 控件一样修改选定单元格中的值。当完成编辑，退出编辑模式后，输入单元格的字符串则显示为相应的字符串值，并在需要的时候对数据库进行更新。

在转换为单元格中显示的值或者与数据库进行更新时，在编辑状态下输入的字符串也将转化为该单元格值的数据类型。此类型是由 `DataGridViewTextBoxColumn` 的 `ValueType` 属性确定的。例如，在编辑状态下，用户可以在列名为“总重”的单元格中输入 1234.5 吨。但是如果该单元格的数据类型被指定为整数型，则在退出编辑模式时，单元格中的值显示为 1234。以上过程主要与两个事件有关，开发人员可以在这两个事件中添加相关处理程序。

`CellFormatting` 事件是在将单元格中的值设置成一定的格式，以便显示给用户以做修改时发生。

`CellParsing` 事件则是在单元格的修改已经完成，用户退出编辑模式时发生。

 **说明：**`DataGridViewTextBoxColumn` 是在 `DataGridView` 控件中一种使用频率最高的列类型。

下面的代码中，通过对这两个事件的处理函数进行定义来说明这两个事件的使用方法。这段代码主要演示的是在双击 DataGridView 控件单元格使其进入编辑模式时，会触发 CellFormatting 事件，此时程序会弹出一个消息框，询问是否确定要对 DataGridViewCell 进行修改。注意，此时用户选择“是”或“否”对程序的继续运行不会造成影响，对于如何使用消息框对程序运行过程进行控制，将在第 16 章中进行详细说明。用户选择后，选中的单元格将进入编辑状态，对单元格中的数据重置后按 Enter 键退出编辑模式，此时将启动 CellParsing 事件。在 CellParsing 事件中添加处理程序，使程序弹出一个消息框，询问是否要提交所做的修改。用户做出选择后，程序将继续执行，单元格退出编辑状态，所做的更改显示为单元格值。

```
//设置一个 boolean 类型的变量
//用以指示是该运行 CellFormatting 事件还是 CellParsing 事件
bool formattingOrparsing = true;
public Form1()
{
    InitializeComponent();
    //初始化开始，将 formattingOrparsing 置为 true，
    //在 CellFormatting 事件发生时，不执行添加的处理程序
    formattingOrparsing = true;
    //使用户不能擅自对 DataGridView 控件添加数据行
    this.dataGridView1.AllowUserToAddRows = false;
    //创建了一个 DataGridViewRow 对象的实例
    DataGridViewRow row1 = new DataGridViewRow();
    object[] values = new object[1] { 1 };
    //对 row1 中的单元格进行了设置
    //参数 dataGridView1 是对 row1 进行设置的模板
    //values 是一个 object 数组，里面包含了对 row1 进行设置的值
    row1.CreateCells(this.dataGridView1, values);
    //将 DataGridViewRow 对象的实例添加到 dataGridView1 中
    this.dataGridView1.Rows.Add(row1);
    //初始化完成，将 formattingOrparsing 置为 false
    //在 CellFormatting 事件发生时，执行添加的处理程序
    formattingOrparsing = false;
}
```

CellFormatting 事件处理函数如下：

```
private void dataGridView1_CellFormatting(
    object sender, DataGridViewCellFormattingEventArgs e)
{
    //判断 formattingOrparsing 是否为 true
    //如果为 true，证明控件现在处于初始化状态或有变化未提交状态，处理程序运行结束
    //如果为 false，则继续运行下面的处理程序
    if (!formattingOrparsing)
    {
        //判断当前选中的单元格的行索引号是否为 0
        //即是否为控件中的第一行数据
        //此参数判定在此事件中意义不大，但在多行应用程序中却经常得到应用，请读者牢记
        if (e.RowIndex == 0)
        {
            //显示消息框消息框
            //消息框中的文本是“确认要进行修改？”
            //标题是“CellFormatting 事件”
        }
    }
}
```



```

        //消息框中显示两个按钮，分别是“是”/“否”
        MessageBox.Show("确认要进行修改?", "CellFormatting 事件", Message-
        BoxButtons.OKCancel);
        //将 formattingOrparsing 置为 true
        //表示单元格中已经有数据进行了重新的编辑，但还没有提交后退出编辑模式
        formattingOrparsing = true;
    }
}
}

```

CellParsing 事件处理函数如下：

```

private void dataGridView1_CellParsing(object sender,
    DataGridViewCellParsingEventArgs e)
{
    //判断 formattingOrparsing 是否为 true
    //如果为 true，则继续运行下面的处理程序
    //如果为 false，证明控件现在编辑结束状态
    if (formattingOrparsing)
    {
        if (e.RowIndex == 0)
        {
            //显示消息框消息框
            //消息框中的文本是“确认要提交修改？”
            //标题是“CellParsing 事件”
            //消息框中显示两个按钮，分别是“是”/“否”
            MessageBox.Show("确认要提交修改?", "CellParsing 事件", Message-
            BoxButtons.OKCancel);
            //将 formattingOrparsing 置为 false
            //表示单元格中已经退出编辑模式
            formattingOrparsing = false;
        }
    }
}

```

程序的运行结果如图 15.20、图 15.21 和图 15.22 所示。

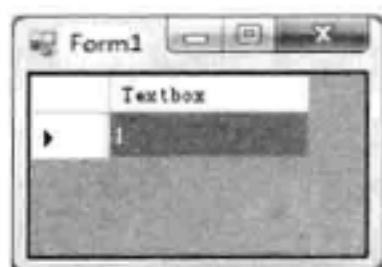


图 15.20 添加列对话框



图 15.21 添加列对话框



图 15.22 column 类型选择

技巧：建议读者在运行此段代码时，将 formattingOrparsing 变量的代码去掉，查看运行效果，并思考为什么会有这种运行效果。

15.4.3 用 DataGridViewCheckBoxColumn 显示二元状态数据列

DataGridViewCheckBoxColumn 在 DataGridView 控件中主要用于显示对某个特定条件是处于打开状态还是处于关闭状态，是一个可表示二元状态的单元格。在 DataGridView 控件中，它用于为用户提供针对某一数据行的特定状态的“是/否”或“真/假”选项。当

DataGridView 控件是可编辑的状态时,如果双击想要编辑的 CheckBox 控件形式的 DataGridViewCell,则相应的单元格就会被激活,显示为一个没有文本标签的 CheckBox 控件的外观,并且可以像 CheckBox 控件一样使用,如图 15.23 所示。在使用 DataGridViewCheckBoxColumn 类时,有以下几个属性需要重点介绍。



图 15.23 DataGridViewCheckBoxColumn 外观

ThreeState 属性用于确定该列所对应的 CheckBox 控件的单元格是支持两种状态还是 3 种状态。如果将 ThreeState 属性设置为 true,则相应的 DataGridViewCheckBoxCell 具有 3 种状态模式,分别为选中状态、未选中状态和不确定状态。不确定状态有时在应用程序中十分有用,例如当用户不希望在复选框中设置默认值时,或用户不确定数据所对应的状态时,均可将 DataGridViewCheckBoxCell 设为不确定状态。如果将 ThreeState 属性设置为 false,则相应的 DataGridViewCheckBoxCell 具有两种状态模式,分别为选中和未选中状态。

当用户完成编辑,退出此单元格时,对单元格所做的选择将会提交给绑定到此控件的数据源。提交的数据由 FalseValue 属性、TrueValue 属性和 IndeterminateValue 属性确定。


FalseValue 属性用于获取或设置 DataGridViewCheckBoxCell 为 false 时(显示为复选框未被选中),向 DataGridView 控件所绑定的数据源传递的相应的数据基础值。

TrueValue 属性用于获取或设置 DataGridViewCheckBoxCell 为 true 时(显示为复选框被选中),向 DataGridView 控件所绑定的数据源传递的相应的数据基础值。

IndeterminateValue 属性用于获取或设置 DataGridViewCheckBoxCell 为 Indeterminate 时,向 DataGridView 控件所绑定的数据源传递的相应的数据基础值。

了解了 DataGridViewCheckBoxColumn 的常用属性,就可以通过代码的方式向 DataGridView 控件中添加 DataGridViewCheckBoxColumn,具体代码如下所示。

```
private void AddCheckBoxColumn()
{
    //创建一个 DataGridViewCheckBoxColumn 对象
    this.Column1 = new System.Windows.Forms.DataGridViewCheckBoxColumn();
    //设置 DataGridViewCheckBoxColumn 对象的相关属性
    this.Column1.FalseValue = "Unchecked";
    this.Column1.HeaderText = "CheckBox";
    this.Column1.IndeterminateValue = "Unknown";
    this.Column1.Name = "Column1";
    this.Column1.ThreeState = true;
    this.Column1.TrueValue = "Checked";
    //将 DataGridViewCheckBoxColumn 对象附加到 dataGridView1 中
    this.dataGridView1.Columns.AddRange(new System.Windows.Forms.DataGridViewColumn[] { this.Column1 });
}
```

 **技巧:** 通过对 FalseValue 属性、TrueValue 属性和 IndeterminateValue 属性进行设置后, DataGridViewCheckBoxCell 中的值就可以像其他数据一样用于存储了。

15.4.4 用 DataGridViewImageColumn 显示图像数据列

DataGridViewImageColumn 是在 DataGridView 控件中用于显示图像的数据列。在应用


程序中, 有时候除了对数据的信息进行浏览, 也希望可以对数据相对应的图片进行浏览。例如人力资源系统中的人员信息所对应的照片。在使用 `DataGridViewImageColumn` 类时, 有以下几个属性需要重点介绍。

`DataGridViewImageColumn` 类的 `Icon` 属性, 是用来设置 `DataGridViewImageColumn` 中单元格的图标, 它是一个 `Icon` 类的对象实例。`DataGridViewImageColumn` 类的 `Image` 属性, 是用来设置 `DataGridViewImageColumn` 中单元格的图像, 它是一个 `Image` 类的对象实例。可以使用以下代码对 `DataGridViewImageColumn` 类的 `Image` 属性进行设置。

```
private void CreateColumns()
{
    DataGridViewImageColumn imageColumn;
    int columnCount = 0;
    while (columnCount < 3)
    {
        Bitmap blankImage = blank;
        imageColumn = new DataGridViewImageColumn();
        //设置 imageColumn 的宽度
        imageColumn.Width = x.Width + 2 * bitmapPadding + 1;
        imageColumn.Image = blankImage;
        //向 dataGridView1 添加图片列
        dataGridView1.Columns.Add(imageColumn);
        columnCount = columnCount + 1;
    }
}
```

`DataGridViewImageColumn` 类的 `ValuesAreIcons` 属性是一个 `boolean` 类型的数值。当 `ValuesAreIcons` 的属性值为 `true` 时, 则在 `DataGridViewImageColumn` 中的单元格中显示的图像由 `Icon` 属性来定义; 当 `ValuesAreIcons` 的属性值为 `false` 时, 则在 `DataGridViewImageColumn` 中的单元格中显示的图像由 `Image` 属性来定义。对 `ValuesAreIcons` 属性的设置如下面的代码所示。

```
private void setImageColumns()
{
    this.dataGridViewImageColumn1.ValuesAreIcons=true;
}
```

 **技巧:** 可以通过对 `ValuesAreIcons` 属性进行设置, 使应用程序可以切换不同的视图。

`DataGridViewImageColumn` 类的 `ImageLayout` 属性是一个 `DataGridViewImageCellLayout` 枚举类型值, 它表示了图形在 `DataGridViewImageColumn` 中的布局方式。`DataGridViewImageCellLayout` 枚举类型的具体成员如表 15.13 所示。

表 15.13 `DataGridViewImageCellLayout` 成员列表

成 员	说 明
NotSet	不对布局方式进行设置
Normal	居中显示图片
Stretch	根据单元格大小进行拉伸
Zoom	在不改变图形长宽比的前提下使图形的长或宽满足单元格的长或宽

可以使用以下代码对 DataGridViewImageColumn 类的 ImageLayout 属性进行设置。代码示例如下所示。


```
private void setImageColumnImageLayout()
{
    this.dataGridViewImageColumn1.ImageLayout=DataGridViewImageCellLayout.Stretch;
}
```

Description 属性是一个字符串类型的值，它用来提供对单元格内图片的描述。对于 Description 属性的设置代码如下所示。

```
private void SetImageDescription ()
{
    //遍历 dataGridView1 中的图片列
    foreach (DataGridViewImageColumn column in dataGridView1.Columns)
    {
        //根据图片列的图片布局，队列设置描述
        if(column.ImageLayout ==DataGridViewImageCellLayout.Stretch)
        {
            column.Description = "Stretched";
        }
        if(column.ImageLayout ==DataGridViewImageCellLayout.Zoom)
        {
            column.Description = "Zoomed";
        }
        if(column.ImageLayout ==DataGridViewImageCellLayout.Normal)
        {
            column.Description = "Normal";
        }
    }
}
```

15.4.5 用 DataGridViewButtonColumn 显示按钮数据列

DataGridViewButtonColumn 是在 DataGridView 控件中用于显示按钮的数据列。在应用程序中，有时除了对数据的信息进行浏览，也希望可以对某些特定数据进行操作。例如单击数据行的按钮单元格，可以在新的窗体中显示此数据行的子数据。

说明：DataGridViewButtonColumn 类的设置方法和 Button 控件相似。

DataGridViewButtonColumn 类的 FlatStyle 属性可以用来设置 DataGridViewButtonColumn 中的单元格的图标，它是一个 FlatStyle 类型的枚举值。可以使用以下代码对 DataGridViewButtonColumn 类的 FlatStyle 属性进行设置。

```
private void SetButtonsFlatStyle()
{
    DataGridViewButtonColumn buttonColumn;
    buttonColumn= new DataGridViewButtonColumn();
    buttonColumn.FlatStyle= FlatStyle.Standard;
}
```


DataGridViewButtonColumn 类的 Text 属性是一个 string 类型的数值，它主要用来设置 DataGridViewButtonColumn 上的按钮单元格的默认文本。对 Text 属性的设置如下面的代码

所示。

```
private void SetButtonsText()
{
    DataGridViewButtonColumn buttonColumn;
    buttonColumn= new DataGridViewButtonColumn();
    buttonColumn.Text="按钮";
}
```

15.4.6 用 DataGridViewComboBoxColumn 显示下拉列表数据列

DataGridViewComboBoxColumn 是在 DataGridView 控件中用于显示下拉列表的数据列。在应用程序中，有时候除了对数据的信息进行浏览，也希望可以对某些特定数据进行输入，当这些数据的内容相对固定且数量较少时，可以通过使用 DataGridViewComboBoxColumn 类，将单元格变成外观与 ComboBox 控件一致的单元格，使用户可以在下拉框中对希望填入的数据进行选择。另外，当希望输入的数据固定为某几项时，也可以使用 DataGridViewComboBoxColumn 类。例如需要用户填入家庭年收入时，就可以通过设置几个范围选项进行选择，以获得较为工整的数据。

说明：DataGridViewComboBoxColumn 类的设置方法和 ComboBox 控件相似。

DataGridViewComboBoxColumn 类的 AutoComplete 属性是一个 boolean 类型的值，可以用来指示在向 DataGridViewComboBoxColumn 上的单元格内输入字符时，此单元格是否能将候选值与输入的值相匹配，如果可以则为 true，否则为 false。对 AutoComplete 属性的设置如下面的代码所示。

```
private void SetComboBoxAutoComplete()
{
    DataGridViewComboBoxColumn comboBoxColumn;
    //候选值可以与输入值相匹配
    comboBoxColumn= new DataGridViewComboBoxColumn();
    comboBoxColumn.AutoComplete= true;
}
```

DataGridViewComboBoxColumn 类的 DisplayMember 属性是一个 string 类型的值，它主要用来获取在 DataGridViewComboBoxColumn 上的单元格中显示的内容。DataGridViewComboBoxColumn 类的 ValueMember 属性的功能与 DisplayMember 属性相类似，ValueMember 属性也是一个 string 类型的数值，它主要用来获取在 DataGridViewComboBoxColumn 上的单元格下拉列表中选中的项的内容。通常情况下，代码设置如下：

```
comboBoxColumn.DisplayMember== comboBoxColumn.ValueMember;
```

也可以使用如下代码给 DisplayMember 属性赋值：

```
private void SettComboBoxDisplayMember ()
{
    comboboxColumn.ValueMember = ColumnName.TitleOfCourtesy.ToString();
}
```

```

        comboBoxColumn.DisplayMember = comboBoxColumn.ValueMember;
    }

```

DisplayStyle 属性则指示了组合框的显示方式，DisplayStyle 属性值是一个 DataGridViewComboBoxDisplayStyle 的枚举类型值。DataGridViewComboBoxDisplayStyle 的主要成员如表 15.14 所示。

表 15.14 DataGridViewComboBoxDisplayStyle 成员列表

成 员	说 明
ComboBox	DataGridViewComboBoxColumn 中的单元格的外观与 ComboBox 外观相似
DropDownButton	DataGridViewComboBoxColumn 中的单元格的右侧带有下拉按钮
Nothing	DataGridViewComboBoxColumn 中的单元格的右侧不带有下拉按钮，外观与普通单元格相同

对 DisplayStyle 属性的设置如下面的代码所示。

```

private void SetComboBoxDisplayStyle()
{
    DataGridViewComboBoxColumn comboBoxColumn;
    comboBoxColumn = new DataGridViewComboBoxColumn();
    //设置组合框的显示方式
    comboBoxColumn.DisplayStyle = DataGridViewComboBoxDisplayStyle.ComboBox;
}

```

DisplayStyleForCurrentCellOnly 属性则是用来表示如果 DataGridView 控件的 SelectedCell 所指示的单元格位于此列时，该单元格是否还按照 DisplayStyle 属性所设置的那样显示单元格。当 DisplayStyleForCurrentCellOnly 属性值为 true 时，该单元格将按照 DisplayStyle 属性所设置的那样显示单元格；当 DisplayStyleForCurrentCellOnly 属性值为 false 时，该单元格将按照 ComboBox 控件的外观显示单元格。对 DisplayStyleForCurrentCellOnly 属性的设置如下面的代码所示。

```

private void SetComboBoxDisplayStyle()
{
    DataGridViewComboBoxColumn comboBoxColumn;
    comboBoxColumn = new DataGridViewComboBoxColumn();
    comboBoxColumn.DisplayStyle = DataGridViewComboBoxDisplayStyle.ComboBox;
    //单元格将按照 ComboBox 控件的外观显示单元格
    comboBoxColumn.DisplayStyleForCurrentCellOnly = false;
}

```

DataGridViewComboBoxColumn 类的 DropDownWidth 属性是一个 int 类型的值，可以用来指示 DataGridViewComboBoxColumn 的单元格的下拉列表的宽度。对 DropDownWidth 属性的设置如下面的代码所示。

```

private void SetComboBoxDropDownWidth()
{
    DataGridViewComboBoxColumn comboBoxColumn;
    comboBoxColumn = new DataGridViewComboBoxColumn();
    comboBoxColumn.DropDownWidth = 200;
}

```


 注意: DropDownWidth 属性还可以在 DataGridViewComboBoxColumn 类的 CellTemplate 属性中进行设置。

DataGridViewComboBoxColumn 类的 Items 属性是一个 DataGridViewComboBoxCell.ObjectCollection 类型的实例, 此属性用来指示在 DataGridViewComboBoxColumn 类的单元格中的下拉列表的选项集合。可以通过调用 DataGridViewComboBoxCell.ObjectCollection 类的方法和属性来对 Items 属性进行添加、查找和获取等操作。对 Items 属性的设置如下面的代码所示。

```
private void SetComboBoxItems()
{
    DataGridViewComboBoxColumn comboBoxColumn;
    comboBoxColumn = new DataGridViewComboBoxColumn();
    comboBoxColumn.Items.AddRange(new string[] { "<1000.", "1000-1500.",
        "1500-2000.", ">2000." });
}
```

MaxDropDownItems 属性标识出了在 DataGridViewComboBoxColumn 类的 Items 属性确定的下拉列表框中项的个数的最大值。当此属性设置后, 如果向 Items 属性里添加的项数大于这个值, 则会抛出异常。对 MaxDropDownItems 属性的设置如下面的代码所示。

```
private void SetComboBoxMaxDropDownItems()
{
    DataGridViewComboBoxColumn comboBoxColumn;
    comboBoxColumn = new DataGridViewComboBoxColumn();
    comboBoxColumn.MaxDropDownItems = 10;
}
```

Sorted 属性是一个 boolean 类型的值, 用来指示是否可以对 DataGridViewComboBoxColumn 类中的单元的组合框的数据进行排序。如果可以排序, Sorted 属性值为 true, 如果不能排序, 则 Sorted 属性值为 false。对 Sorted 属性的设置如下面的代码所示。

```
private void SetComboBoxSorted()
{
    DataGridViewComboBoxColumn comboBoxColumn;
    comboBoxColumn = new DataGridViewComboBoxColumn();
    comboBoxColumn.Sorted = true;
}
```

15.4.7 用 DataGridViewLinkColumn 显示超链接数据列

DataGridViewLinkColumn 是在 DataGridView 控件中用于显示超链接文本的数据列。在应用程序中, 有时希望在单击某个数据时, 可以自动打开此数据 (多数为网址) 制定的超链接地址。这时候可以使用 DataGridViewLinkColumn 类。该类提供了多个属性, 用于在单击链接时、单击链接之前和之后修改链接的外观。下面将对其进行逐一介绍。

 说明: DataGridViewLinkColumn 类的设置方法和 LinkLabel 控件相似。

DataGridViewLinkColumn 类的 LinkColor 属性用来设置单元格中的链接未被选中时的颜色, ActiveLinkColor 属性用来设置单元格中的链接被选中时的颜色, VisitedLinkColor

属性用来设置单元格中的链接被访问后的颜色。通过以上 3 种属性，则可以对 DataGridViewLinkColumn 类的 3 种不同状态的颜色进行设置，设置的具体代码如下所示。

```
private void SetLinksColor()
{
    DataGridViewLinkColumn linkColumn;
    linkColumn= new DataGridViewLinkColumn();
    //设置单元格中的链接被选中时的颜色
    linkColumn.ActiveLinkColor= Color.Green;
    //设置单元格中的链接未被选中时的颜色
    linkColumn.LinkColor= Color.Red;
    //设置单元格中的链接被访问后的颜色
    linkColumn.VisitedLinkColor=Color.Blue;
}
```

DataGridViewLinkColumn 类的 UseColumnTextForLinkValue 属性是一个 boolean 类型的值，它主要用来指示在 DataGridViewLinkColumn 上的单元格文本中的链接是否显示。如果显示则为 true，否则为 false。对 UseColumnTextForLinkValue 属性的设置如下面的代码所示。

```
private void SetUseColumnTextForLinkValue()
{
    DataGridViewLinkColumn linkColumn;
    linkColumn= new DataGridViewLinkColumn();
    linkColumn.UseColumnTextForLinkValue=true;
}
```

DataGridViewLinkColumn 类的 TrackVisitedState 属性是一个 boolean 类型的值，它主要用来指示在 DataGridViewLinkColumn 上的单元格中的链接被访问后，是否显示可以更改颜色。如果显示则为 true，否则为 false。如果 TrackVisitedState 的属性值为 false，则 ActiveLinkColor 属性和 VisitedLinkColor 属性的设置将没有任何意义。对 TrackVisitedState 属性的设置如下面的代码所示。

```
private void SetTrackVisitedState()
{
    DataGridViewLinkColumn linkColumn;
    linkColumn= new DataGridViewLinkColumn();
    //单元格中的链接被访问后，可以更改颜色
    linkColumn.TrackVisitedState=true;
}
```

LinkBehavior 属性用来设置 DataGridViewLinkColumn 类的单元格中的超链接的行为。LinkBehavior 属性值是一个 LinkBehavior 枚举类型的值。LinkBehavior 的主要成员如表 15.15 所示。

表 15.15 LinkBehavior成员列表

成 员	说 明
AlwaysUnderline	文本中的超链接下始终带有下列线
HoverUnderline	只有在鼠标划过时，文本中的超链接才带有下列线
NeverUnderline	文本中的超链接不带有下列线
SystemDefault	超链接形式设置与系统设置一致

对 LinkBehavior 属性的设置如下面的代码所示。

```
private void SetLinkBehavior()  
{  
    DataGridViewLinkColumn linkColumn;  
    linkColumn= new DataGridViewLinkColumn();  
    //设置单元格中的超链接的行为  
    linkColumn.LinkBehavior=AlwaysUnderline;  
}
```

15.5 本章总结

在本章中，主要对 DataGridView 控件的一些常用属性和方法进行了介绍，并分析了 DataGridView 相关类之间的关系。DataGridView 控件更强大的功能是可以与后台数据库结合使用，自动生成 DataGridView 控件结构，这些内容将在后面的章节中介绍。

15.6 实战练习

1. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在 Form1 中添加一个 DataGridView 控件，在开发环境中通过可视化方式为 DataGridView 控件添加 5 列，并设置各列分别显示文本、复选框、下拉列表框、图像、按钮等类型的数据。
2. 接第 1 题，通过编写代码的方式冻结 DataGridView 控件中的第 1 列。
3. 接第 1 题，通过编写代码的方式，分别为表头和表格设置不同的单元格边框样式。

第 16 章 通用对话框

在 Windows 系统中提供了一些通用对话框，如打印设置对话框、页面设置对话框、查找对话框、文件对话框、字体对话框和颜色对话框等。这些对话框同样也是在 Windows 应用系统编程中使用频率很高的一些对话框。.NET 框架通过将这些常用的对话框封装成组件，使开发人员在需要用这些对话框时，可以将其作为资源文件添加到窗体中，并通过程序的设计，使这些对话框在需要的时候弹出，然后读取相应的属性获得用户在对话框中所做的选择。

16.1 消息对话框

本节将对消息对话框进行详细的介绍。

16.1.1 用消息对话框显示信息

消息对话框是一个特殊的 Windows 应用程序通用对话框，它主要用来向用户显示提示信息、警告信息等相关信息。在 Windows 应用程序中，通用对话框是一个很常用的对话框，其在运行过程中的外观如图 16.1 所示。

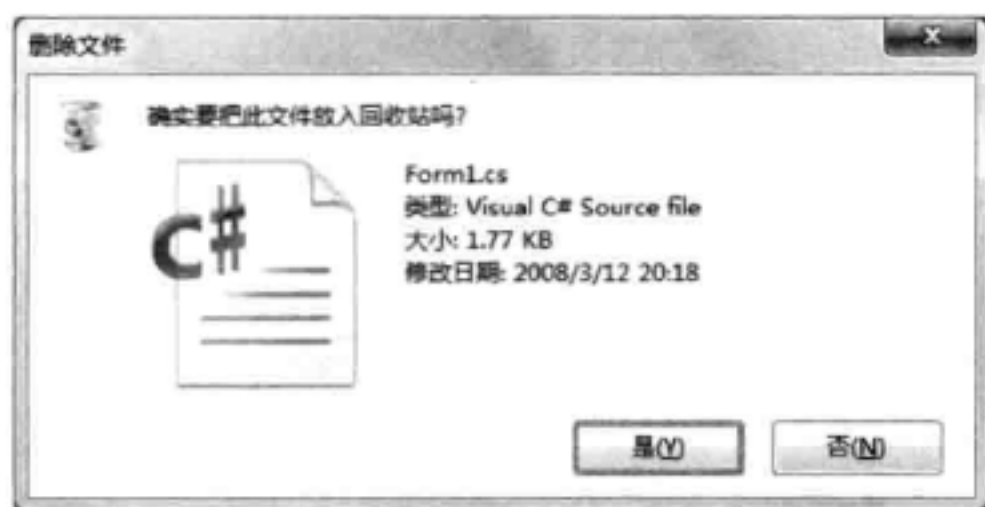



图 16.1 消息对话框

在使用消息对话框时，用户根据显示的信息进行相应的判断，并单击不同的按钮进行选择，程序代码通过获取用户的选择来控制程序运行的流程。

在 .NET 框架中，消息对话框的命名空间为 `System.Windows.Forms`。在 Windows 应用程序开发中，开发人员可以通过使用 `System.Windows.Forms.MessageBox` 类来显示一个消息对话框。

 **注意：** MessageBox 类是无法实例化的，读者可以直接调用 MessageBox 类的静态方法 Show()来显示消息对话框。

16.1.2 该怎样显示消息对话框

在使用 MessageBox 类的 Show()方法时，根据传入的参数不同，所弹出的消息对话框的外观也会有所区别，下面将通过具体的例子对 MessageBox 的 Show()方法进行介绍。

在下面的程序中，将通过代码显示一个对话框，在对话框中显示一条指定的文本，这种方法主要用来向用户显示提示信息。具体代码如下：

```
private void PerformMessageBox1()
{
    //传入字符串参数，显示在对话框中
    MessageBox.Show("文件复制已完成");
}
```

将上面的代码添加到执行复制功能的单线程 Windows 应用程序的函数结尾处，在复制任务结束时，将弹出如图 16.2 所示的消息对话框，用于提示用户文件复制任务已经完成。用户可以通过单击消息框中的“确定”按钮来关闭消息对话框，关闭消息对话框后，程序将继续运行。

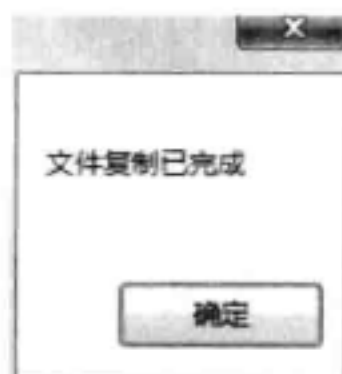



图 16.2 消息对话框

 **说明：** 应用程序中弹出的消息对话框与其他通用对话框相同，大小是不能进行调整的（无论是开发人员在开发过程中通过程序进行设置，还是用户在运行过程中进行调整，都是不被允许的）。弹出消息框时，应用程序将根据消息对话框中的内容自动调整消息对话框的大小。

在图 16.2 中显示的消息对话框是一个运行界面最简单的消息对话框。在通常情况下，Windows 应用程序所弹出的消息对话框，要比这个对话框相对复杂，而且也可以提供相应的具体功能。例如显示提示图标、消息框标题等。在下面的例子中，将分别就这些功能的添加进行介绍。

在下面的代码中，将演示如何生成一个消息对话框，在这个消息对话框中，将显示指定的字符串作为提示信息，同时在消息对话框的标题栏中显示文本，这个所显示的文本通常用于说明消息对话框的用途。

```
private void PerformMessageBox1()
{
    //第一个字符串参数将显示在对话框中
    //第二个字符串参数将显示在对话框标题栏中
    MessageBox.Show("文件复制已完成", "提示: ");
}
```

同样，可以将上面的代码添加到执行复制功能的单线程 Windows 应用程序的函数结尾处，在复制任务结束时，将弹出如图 16.3 所示的消息对话框，用于提示用户文件复制任务已经完成。读者可以仔细观察这个新生成的消息对话框与图 16.2 所示的消息对话框的区别，在这个新生成的对话框的标题中，显示了“提示：”字符串，用以说明这个消息对话框

框的功能。同样，用户可以通过单击“确定”按钮关闭消息对话框，使程序继续运行。

消息对话框的弹出位置通常是在显示屏的中间、所有窗体的最前方，以方便用户很容易就能发现。但是在有的应用程序中，用户希望弹出的消息对话框显示在指定的窗体上方，也就是在弹出消息对话框的窗体的上方。这时可以使用以下代码完成此功能：

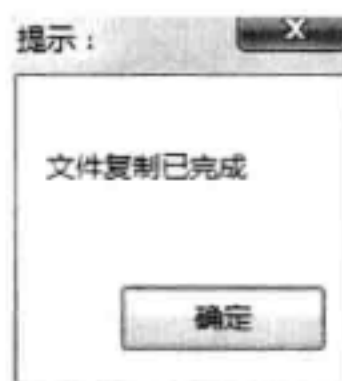


图 16.3 消息对话框

```
private void PerformMessageBox2()
{
    //第一个参数用来指示消息对话框所显示在其前的窗体
    //第二个字符串参数将显示在对话框中
    //第三个字符串参数将显示在对话框标题栏中
    MessageBox.Show(this, "文件复制已完成", "提示: ");
}
```

在上面的方法中，与前一段程序的主要区别在于，Show()方法中增添的一个参数用于指示消息对话框所处于其前的窗体。通常情况下，当需要指定这一窗体时，都是希望消息对话框显示在弹出这一对话框的窗体的前方，那么在程序实现中，只需要在Show()方法的其他参数前面加上this这个指示父窗体的参数即可。无论在Show()方法中，还有多少其他指定的参数，都可如此进行设置。


16.1.3 消息对话框能显示哪些按键

在前面的3个示例程序中，弹出的消息对话框上都有一个确定按钮，用于关闭当前的消息对话框，使程序继续运行，但是有时候，应用程序所弹出的消息对话框将会显示一个询问信息，希望用户做出选择，并根据用户做出的选择继续程序的运行，这个时候，就要求所弹出的对话框上显示多个按钮，以使用户做出不同的选择。如图16.1所示的消息对话框中，就有两个按钮，两个按钮相应的显示文本是“是(Y)”与“否(N)”。当用户单击“是”按钮时，指定的文件将被删除；当用户单击“否”按钮时，将会取消这一操作，文件不会被删除。

在下面的代码中，将假设程序在执行上传文件的操作。在操作执行过程中，程序发现有一个上传的文件处于编辑状态无法完成上传功能，此时程序将抛出一个异常。在异常处理函数中，添加下面的方法，使程序弹出一个消息对话框，向用户提示异常。

```
private void PerformMessageBox3()
{
    //第一个字符串参数将显示在对话框中
    //第二个字符串参数将显示在对话框标题栏中
    //第三个参数是 MessageBoxButtons 枚举类型的枚举值，用来指示显示在消息对话框中的按钮
    MessageBox.Show("文件“第16章 通用对话框”未关闭，不能执行上传操作", "错误",
        MessageBoxButtons.AbortRetryIgnore);
}
```


将上文中定义的方法添加到一个 Windows 应用程序的上传文件的代码异常处理函数中，并对抛出的异常的类型进行判断，当异常的引发原因是上传的文件处于编辑状态时，将弹出如图 16.4 所示的消息对话框。

 **技巧：**开发人员可以通过设置 `MessageBoxButtons` 枚举值来设置消息对话框上的按钮。

在弹出的消息对话框中，将显示 3 个按钮，其文本分别为“终止”、“重试”和“取消”，用户可根据不同的需求对按钮进行单击，根据用户的不同响应，程序将执行不同的操作。

观察上面的代码，读者可以发现在 `MessageBox` 的 `Show()` 方法中共有 3 个参数。其中第 3 个参数是一个 `MessageBoxButtons` 枚举类型的枚举值，主要用来指示在消息对话框中所显示的按钮。`MessageBoxButtons` 的主要成员如表 16.1 所示。

表 16.1 `MessageBoxButtons` 的主要成员

成员名称	说 明
OK	在消息对话框中将显示“确定”按钮，如图 16.5 所示
OKCancel	在消息对话框中将显示“确定”和“取消”按钮，如图 16.6 所示
YesNo	在消息对话框中将显示“是”和“否”按钮，如图 16.7 所示
YesNoCancel	在消息对话框中将显示“是”，“否”和“取消”按钮，如图 16.8 所示
RetryCancel	在消息对话框中将显示“重试”和“取消”按钮，如图 16.9 所示
AbortRetryIgnore	在消息对话框中将显示“终止”、“重试”和“忽略”按钮，如图 16.10 所示



图 16.5 消息对话框



图 16.6 消息对话框



图 16.7 消息对话框



图 16.8 消息对话框



图 16.9 消息对话框




图 16.10 消息对话框

16.1.4 用户按了哪个按钮

正如前面所讲解的，当消息对话框中的按钮数量大于一个的时候，Windows 应用程序可以根据用户所单击的按钮进行不同的响应。那么程序是如何判断用户的响应呢？在应用

程序中,当用户做出选择后,消息框会自动关闭,并返回一个 DialogResult 枚举类型的值,指示用户所做的选择。具体使用方法如下所示。

```
private void CheckUserName()
{
    //当用户名文本框中的字符串长度为 0 时执行
    if(this.textBox1.Text.Length == 0)
    {
        string message = "没有输入用户名,是否取消当前操作?";
        string caption = "输入错误";
        //消息对话框中将显示“是”与“否”两个按钮
        MessageBoxButtons buttons = MessageBoxButtons.YesNo;
        //用于捕获消息对话框的返回值
        DialogResult result;
        //第一个字符串参数将显示在对话框中
        //第二个字符串参数将显示在对话框标题栏中
        //第三个参数是 MessageBoxButtons 枚举类型的枚举值,用来指示显示在消息对话框
        中的按钮
        result = MessageBox.Show(message, caption, buttons);
        //当用户单击“是”按钮时,关闭窗口体
        if(result == DialogResult.Yes)
        {
            this.Close();
        }
    }
}
```

 **说明:** 程序通过判断消息对话框返回的 DialogResult 值来判断用户的响应。

在上面的示例程序中,当没有在用户名文本框中输入用户名时,将会弹出一个消息对话框,提示用户没有输入用户名,是否取消当前操作。当用户做出选择时,则使用 DialogResult 类的实例来捕获这一操作的返回值,对返回值进行判断,如果返回值为 DialogResult.Yes,则退出程序,关闭窗口体。具体代码如下所示。

```
//用于捕获消息对话框的返回值
DialogResult result;
//第一个字符串参数将显示在对话框中
//第二个字符串参数将显示在对话框标题栏中
//第三个参数是 MessageBoxButtons 枚举类型的枚举值,用来指示显示在消息对话框中的按钮
result = MessageBox.Show(message, caption, buttons);
//当用户单击“是”按钮时,关闭窗口体
if(result == DialogResult.Yes)
{
    this.Close();
}
```

在上面的例子中,消息对话框返回的是 DialogResult.Yes 枚举值,此枚举值代表当消息对话框中显示为“是”按钮时,用户通过单击“是”按钮关闭消息对话框。当用户单击

消息对话框中的其他按钮时，将返回其他的 DialogResult 枚举值。DialogResult 的具体主要成员如表 16.2 所示。

表 16.2 DialogResult 的主要成员

成 员 名 称	说 明
None	对话框还处于运行状态中
OK	点击对话框中的“确定”按钮后的返回值
Cancel	点击对话框中的“取消”按钮后的返回值
Abort	点击对话框中的“终止”按钮后的返回值
Retry	点击对话框中的“重试”按钮后的返回值
Ignore	点击对话框中的“忽略”按钮后的返回值
Yes	点击对话框中的“是”按钮后的返回值
No	点击对话框中的“否”按钮后的返回值

16.1.5 消息对话框可显示哪些图标

在 16.1.4 节所举的例子中，程序所弹出的消息对话框中包含了标题栏信息，窗体主体部分的提示信息 and 用户选择按钮。但是，在有的消息对话框中，还包含了相应的图标，并以明显的图标显示窗体的用途。这些图标有的用于提示错误，有的用于显示警告，根据图标的内容不同，用户可以清楚地知道应用程序目前所处的状态。

在下面的示例代码中，将显示一个消息对话框用以在用户想要退出程序时进行询问。

```
public void ExitApplication()
{
    //第一个字符串参数将显示在对话框中
    //第二个字符串参数将显示在对话框标题栏中
    //第三个参数是 MessageBoxButtons 枚举类型的枚举值，用来指示显示在消息对话框中的按钮
    //第四个参数是 MessageBoxIcon 枚举类型的枚举值，用来指示显示在消息对话框中的图标
    if (MessageBox.Show ("确定想退出吗", "应用程序",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question)
        == DialogResult.Yes)
    {
        Application.Exit();
    }
}
```

 **技巧：**开发人员通过设置 MessageBoxIcon 枚举值来设置消息对话框上的图标。

将上面的方法添加到应用程序的退出代码中，在程序退出时，会显示如图 16.11 所示的消息框。

读者可以发现，在此消息框中，有一个蓝色底色的问号形状的图案。这个图案是由下列语句中的第 4 个参数 MessageBoxIcon.Question 所指示出来的。

```
MessageBox.Show ("确定想退出吗", "应用程序",
    MessageBoxButtons.YesNo, MessageBoxIcon.Question)
```

MessageBoxIcon.Question 是一个 MessageBoxIcon 枚举类



图 16.11 消息对话框

型的值，在 `MessageBox` 的 `Show()` 方法中，主要用于指示消息对话框中的图标。`MessageBoxIcon` 的主要成员如表 16.3 所示。

表 16.3 `MessageBoxIcon` 的主要成员

成员名称	说 明
<code>None</code>	消息对话框中没有图标显示，如图 16.10 所示
<code>Informationm</code>	当消息对话框用于显示提示信息时可使用此值，图标如图 16.14 所示
<code>Question</code>	当消息对话框询问时可使用此值，图标如图 16.11 所示
<code>Warning</code>	当消息对话框用于显示警告信息时可使用此值，图标如图 16.12 所示
<code>Error</code>	当消息对话框用于显示错误信息时可使用此值，图标如图 16.13 所示
<code>Stop</code>	当消息对话框用于提示程序意外停止时可使用此值，图标如图 16.13 所示
<code>Exclamation</code>	当希望消息对话框显示一个感叹号时可使用此值，图标如图 16.12 所示
<code>Hand</code>	当希望消息对话框显示一个叉形符号时可使用此值，图标如图 16.13 所示
<code>Asterisk</code>	当希望消息对话框显示一个提示型感叹号时可使用此值，图标如图 16.14 所示



图 16.12 消息对话框



图 16.13 消息对话框




图 16.14 消息对话框

16.1.6 消息对话框编程示例

在 16.1.2 节所讲的例子中，给出了在使用消息对话框时，常用的 `Show()` 方法的参数。除了上文所提到的功能外，消息对话框还具有很多的用户订制性。例如：

- ☐ 开发人员可以指定弹出的消息对话框中的默认选定按钮。
- ☐ 开发人员可以指定弹出的消息对话框中的显示方式。
- ☐ 开发人员可以指定弹出的消息对话框中是否显示帮助按钮。以及定义当显示消息按钮时，用户单击消息按钮后窗体的行为。

 **说明：**关于其他定制性的具体说明，请读者参见 MSDN。

在下面的代码中，将对上面所介绍的消息对话框的其他功能进行演示。

```
//当单击窗体上的名为 button1 的按钮时，将调用 ShowMessageBox() 方法
private void button1_Click(object sender, EventArgs e)
{
    DialogResult dr;
    //ShowMessageBox() 方法将返回一个 DialogResult 类型的返回值
    dr= ShowMessageBox();
    //当此返回值的类型为 DialogResult.OK 时，窗体关闭
    if (dr == DialogResult.OK)
    {
        this.Close();
    }
}
```



```

//在下面的代码中, 将显示一个具有帮助按钮的消息对话框
//在此消息对话框中, 当单击帮助按钮时会触发 HelpRequested 事件
//当 HelpRequested 事件被触发时, 将弹出一个用户自定的帮助窗体
//ShowMessageBox( )方法将返回一个 DialogResult 类型的返回值
private DialogResult ShowMessageBox()
{
    //注册 Form1_HelpRequested 事件
    this.HelpRequested += new System.Windows.Forms.HelpEventHandler(this.
        Form1_HelpRequested);

    //显示一个消息对话框
    //第一个字符串参数将显示在对话框中
    //第二个字符串参数将显示在对话框标题栏中
    //第三个参数是 MessageBoxButtons 枚举类型的枚举值, 用来指示显示在消息对话框中的
        按钮
    //第四个参数是 MessageBoxIcon 枚举类型的枚举值, 用来指示显示在消息对话框中的图标
    //第五个参数是 MessageBoxDefaultButton 枚举类型的枚举值, 用来指示在消息对话框
        中的默认选定的按钮
    //第六个参数是 int 类型的值, 用来指示消息对话框中显示形式
    //第七个参数是 bool 类型的值, 用来指示在消息对话框中是否显示帮助按钮
    DialogResult r = MessageBox.Show("您确定要退出应用程序吗?",
        "应用程序",
        MessageBoxButtons.OK,
        MessageBoxIcon.Question,
        MessageBoxDefaultButton.Button1,
        0, true);

    //将 this.Form1_HelpRequested 从 this.HelpRequested 的调用从列表中清除
    this.HelpRequested -= new System.Windows.Forms.HelpEventHandler(this.
        Form1_HelpRequested);

    //返回消息对话框的运行结果
    return r;
}

//当用户单击了消息框中的“帮助”按钮时, 将调用 Form1_HelpRequested 的处理方法
//此方法将自动生成一个用于显示帮助信息的窗体
private void Form1_HelpRequested(System.Object sender, System.Windows.
    Forms.HelpEventArgs hlpevent)
{
    //实例化一个 Form() 类, 用于显示帮助信息
    Form helpForm = new Form();

    //设置这个窗体实例的相关属性
    helpForm.StartPosition = FormStartPosition.Manual;
    helpForm.Size = new Size(600, 200);
    //实例化的窗体将在主窗体旁边显示
    helpForm.DesktopLocation = new Point(this.DesktopBounds.X + this.Size.
        Width, this.DesktopBounds.Top);
    helpForm.Text = "帮助信息";

    //实例化一个窗体中的 Label 控件, 用于显示帮助信息
    Label helpLabel = new Label();
    //将 label 控件添加到窗体中

```



```

helpForm.Controls.Add(helpLabel);
helpLabel.Dock = DockStyle.Fill;
string helpText= "//在下面的代码中, 将显示一个具有帮助按钮的消息对话框\r\n"
helpText=helpText+"//在此消息对话框中, 当单击帮助按钮时会触发 HelpRequested
事件\r\n";
helpText= helpText +"//当 HelpRequested 事件被触发时, 将弹出一个用户自定的帮
助窗体\r\n";
helpText= helpText +"//ShowMessageBox( )方法将返回一个 DialogResult 类型
的返回值\r\n";
helpText= helpText + "private DialogResult ShowMessageBox()\r\n";
helpText= helpText + "{\r\n";
helpText= helpText + "    //注册 Form1_HelpRequested 事件\r\n";
helpText= helpText + "    this.HelpRequested += new
System.Windows.Forms.HelpEventHandler(this.Form1_HelpRequested);\r\n";
helpText= helpText + "    //显示一个消息对话框\r\n";
helpText= helpText + "    DialogResult r = MessageBox.Show(\"您确定要退
出应用程序吗? \",\r\n";
helpText= helpText + "    \"应用程序\",\r\n";
helpText= helpText + "    MessageBoxButtons.OK,\r\n";
helpText= helpText + "    MessageBoxIcon.Question,\r\n";

helpLabel.Text = helpText;
//将自定义的帮助信息窗体显示
this.AddOwnedForm(helpForm);
//显示自定义的帮助信息窗体
helpForm.Show();
hlpevent.Handled = true;
}

```

运行上面的程序, 将显示如图 16.15 所示的消息对话框。

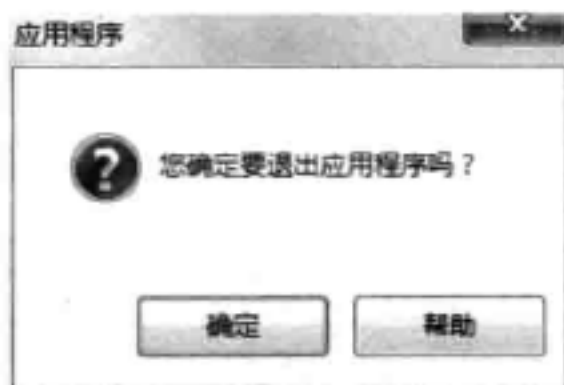


图 16.15 消息对话框

在此消息对话框中, 除显示了 MessageBoxButtons.OK 所定义的“确定”按钮外, 还显示了一个“帮助”按钮。单击这个按钮将触发 HelpRequested 事件, 程序将自动生成一个帮助信息窗体, 显示相应的帮助信息。在这里将显示程序的部分代码, 如图 16.16 所示。



图 16.16 自定义帮助信息窗体

16.2 文件对话框

文件对话框是什么？就是用户在使用软件时，打开、保存或者查找文件所弹出的对话框。在.NET 中用于显示文件对话框的组件主要有用于打开文件的 OpenFileDialog 组件、保存文件的 SaveFileDialog 组件和查找指定文件夹的 FolderBrowserDialog 组件等几种。

16.2.1 认识 OpenFileDialog 组件

“打开”文件对话框是 Windows 系统中用于打开文件的通用对话框。可以通过实例化 System.Windows.Forms.OpenFileDialog 类获得一个打开文件通用对话框。如图 16.17 所示就是一个通用的“打开”文件对话框。



图 16.17 “打开”文件对话框


OpenFileDialog 类的命名空间为 System.Windows.Forms，它在 Visual Studio 工具箱中的图示如图 16.18 所示，将 OpenFileDialog 组件拖曳到窗体，可看到 OpenFileDialog 组件放置在窗体下方的组件托盘区中，如图 16.19 所示。下一步，可通过对 OpenFileDialog 的属性进行设置，使其达到应用程序的具体需求。



图 16.18 OpenFileDialog 组件



图 16.19 OpenFileDialog 组件

说明：在 Windows 应用程序中，通常都是在单击某个按钮或者菜单上的相应选项后才会打开“打开”文件对话框。

16.2.2 显示“打开文件”对话框

在下面的按钮 click 处理事件中，将打开一个文件对话框，并将所选择的文件名字用消息对话框显示出来，代码如下。

```
private void button1_Click(object sender, EventArgs e)
{
    //弹出“打开文件对话框”
    DialogResult dr = this.openFileDialog1.ShowDialog();
    //如果用户单击“确定”按钮
    if (dr == DialogResult.OK)
    {
        //弹出消息对话框，显示用户选择的文件的文件名
        MessageBox.Show(openFileDialog1.FileName, "选择打开的文件");
    }
}
```

在这个例子中，程序使用了 ShowDialog() 方法，这个方法将在屏幕上显示一个对话框，并返回一个 DialogResult 枚举类型，用于说明用户如何关闭对话框。将程序编译运行后，单击窗体上的 button1 按钮，将弹出如图 16.17 所示的“打开”文件对话框，选择 c-sharp.txt 文件后单击“打开”按钮，程序将对返回的 DialogResult 枚举类型值进行判断，如果返回值为 DialogResult.OK，则弹出如图 16.20 所示的消息对话框，指示用户选择文件的路径。



图 16.20 运行结果

16.2.3 打开快捷方式引用的文件

通常情况下，用户即使指定的是某个文件的快捷方式，但是程序打开的依旧是快捷方式文件所指定的文件，当用户使用“打开文件对话框”打开文件时，将浏览到文件的“快捷方式”，将显示如图 16.21 所示提示对话框。示例代码如下。

```
private void button2_Click(object sender, EventArgs e)
{
    //当用户指定的程序是某个文件的快捷方式时，打开的依旧是快捷方式文件所指定文件
    openFileDialog1.DereferenceLinks = false;
    DialogResult dr = this.openFileDialog1.ShowDialog();
    //用户单击确定按钮
    if (dr == DialogResult.OK)
    {
        //弹出消息对话框，显示用户选择的文件名
        MessageBox.Show(openFileDialog1.FileName, "选择打开的文件");
    }
}
```



图 16.21 - 运行结果

从上面的例子可以看出,程序通过读取 OpenFileDialog 组件的 FileName 属性来获取用户所选择文件的路径。

说明:通常情况下,在 Windows 应用程序开发中,使用 OpenFileDialog 组件来打开“打开”文件对话框是为了让用户可以方便地指定希望执行操作的文件,而不是仅仅为了读取用户指定文件的名字。

在下面的例子中,将通过弹出一个“打开”文件对话框让用户指定相应的文本文件,并使用“记事本”打开这个文本文件。

```
private void button3_Click(object sender, EventArgs e)
{
    DialogResult dr = this.openFileDialog1.ShowDialog();
    if (dr == DialogResult.OK)
    {
        //启动“记事本”进程,打开指定的文件
        System.Diagnostics.Process.Start("notepad", openFileDialog1.FileName);
    }
}
```

在上面例子里,将 OpenFileDialog 组件的 FileName 属性设置为 Process.Start()方法的一个参数,使用“记事本”打开用户选定的文件,如图 16.22 所示。

在上面例子中,详细阐述了如何使用 OpenFileDialog 组件打开相应的文件。但是,在 Windows 应用程序开发中,用户的需求往往没有这么简单,开发人员还需要对“打开”文件对话框进行各种限制,这些限制通常也是通过对 OpenFileDialog 组件的属性设置来完成的。对于这些属性的使用,将在下一节详细说明。

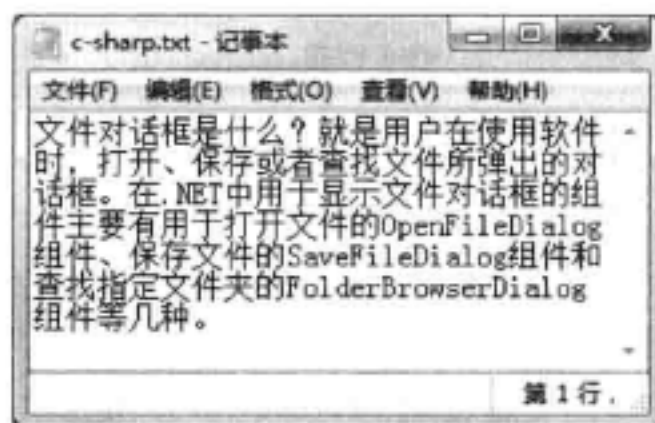


图 16.22 运行结果

16.2.4 同时选择多个文件

当用户希望可以在“打开”文件对话框中进行多选时,需要对 OpenFileDialog 组件的 Multiselect 属性进行设置,当 Multiselect 属性为 true 时,用户可以通过 Ctrl 键或 Shift 键对文件进行多选。对于多选的结果,用户可以通过读取 FileNames 属性获得用户选择的所有文件名。

技巧:当 Multiselect 值为 false 时,程序可以通过读取 FileName 属性来获取所选文件名;当 Multiselect 值为 true 时,程序推荐通过遍历 FileNames 属性来获取所选文件名。

在下例中,用户可在“打开”文件对话框中进行多项选择,并使用消息对话框给出选中文件的文件名。

```
private void button4_Click(object sender, EventArgs e)
{
    //指定在“打开”文件对话框中可进行多项选择
    openFileDialog1.Multiselect = true;
    DialogResult dr = this.openFileDialog1.ShowDialog();
}
```

```

//当用户单击“确定”按钮
if (dr == DialogResult.OK)
{
    //获取用户选择的文件的文件名
    string[] files = this.openFileDialog1.FileNames;
    string fileName = "";
    //遍历这些文件名
    foreach (string file in files)
    {
        fileName = fileName+file+"\r\n";
    }
    //弹出消息对话框，显示用户选择文件的文件夹
    MessageBox.Show(fileName, "选择的文件");
}
}

```

将上面的程序编译运行，单击窗体上的 button4 按钮时，用户将打开相应的可进行多选的“打开”文件对话框。在这个对话框中，通过 Ctrl 键或 Shift 键对多个文件进行选择，读者可以发现，在对话框下方的“文件名”文本框中，将出现用户所选择的所有文件的文件名，如图 16.23 所示。

单击“打开”按钮，将关闭对话框，读取 OpenFileDialog 组件的 FileNames 属性，获得用户所选择文件的文件名列表，并将其显示在消息对话框中，如图 16.24 所示。




图 16.23 “打开”文件对话框



图 16.24 运行结果

16.2.5 设置可打开的文件类型

观察图 16.23 显示的对话框，可看出在该窗体中没有选择文件类型的地方，所以在该对话框中，列举出了指定文件夹中的所有文件及文件夹。如果需要对文件类型进行限制，可以通过定义 OpenFileDialog 组件的 Filter 属性来完成。

说明：“打开”文件对话框中可以设置希望浏览的多种文件类型。

在下面的程序示例中，通过对 OpenFileDialog 组件的 Filter 进行设置，使程序的运行结果如图 16.25 所示。



图 16.25 “打开”文件对话框

```
private void button5_Click(object sender, EventArgs e)
{
    //指定在“打开”文件对话框中可以显示何种类型的文件
    openFileDialog1.Filter = "文本文件(*.txt)|*.txt|所有文件|*.*";
    DialogResult dr = this.openFileDialog1.ShowDialog();
    if (dr == DialogResult.OK)
    {
        MessageBox.Show(openFileDialog1.FileName, "选择的文件");
    }
}
```

在图 16.25 中，程序的文件类型文本框中，默认值为“文本文件(*.txt)”。在打开文件对话框的列表视图中，也只列举了此文件夹中的文本类型文件和文件夹类型的文件。单击文件夹类型文本框旁的向下箭头按钮，打开相应的下拉列表框。在列表中，共有两种文件类型可供选择：

- ☐ 文本文件(*.txt)。
- ☐ 所有文件(*.*)。

程序通过以下代码完成此项功能：

```
openFileDialog1.Filter = "文本文件(*.txt)|*.txt|所有文件|*.*";
```

用户单击“确定”按钮后，将关闭对话框，然后会弹出一个消息对话框显示用户选择打开的文件的文件名。

前面所举的例子中，在弹出“打开”文件对话框时，开发人员并没有在代码中指定 OpenFileDialog 组件所打开的文件夹。在这种情况下，程序打开的文件夹，是系统默认的文件夹，通常是应用程序所在的文件夹。如果用户希望可以在弹出“打开”文件对话框时，就打开某一固定的文件夹，那么开发人员可以通过设置程序的 InitialDirectory 属性来完成，

具体代码如下：

```
private void button6_Click(object sender, EventArgs e)
{
    //指定在“打开”文件对话框中可以显示何种类型的文件
    openFileDialog1.Filter = "文本文件(*.txt)|*.txt|所有文件|*.*";
    //指定在“打开”文件对话框弹出时，所显示的文件夹
    openFileDialog1.InitialDirectory = @"C:\WINDOWS";
    //指定在“打开”文件对话框弹出时，所选定的文件
    openFileDialog1.FileName = "c-sharp.txt";
    //指定在“打开”文件对话框弹出时，此对话框的标题
    openFileDialog1.Title = "打开 WINDOWS 文件夹";
    DialogResult dr = this.openFileDialog1.ShowDialog();
    if (dr == DialogResult.OK)
    {
        MessageBox.Show(openFileDialog1.FileName, "选择的文件");
    }
}
```

编译并运行程序，程序的运行结果如图 16.26 所示。



图 16.26 “打开 WINDOWS 文件夹”对话框


16.2.6 设置“打开”文件对话框的外观

在这段程序中，程序通过以下语句对 `InitialDirectory` 属性赋初值，使窗体打开的文件夹变为“C:\WINDOWS”。

```
openFileDialog1.InitialDirectory = @"C:\WINDOWS";
```

`OpenFileDialog` 组件的 `FileName` 属性不仅能获取用户最终选择的文件，也可以通过此属性设置窗体打开时默认选定的文件，具体代码如下所示。

```
openFileDialog1.FileName = "c-sharp.txt";
```


 **技巧：** OpenFileDialog 组件的 FileName 属性不仅能获取用户最终选择的文件，也可以通过此属性来设置窗体打开时默认选定的文件。

在图 16.26 中，打开文件对话框的标题变为了“打开 WINDOWS 文件夹”，这是在程序里通过对 OpenFileDialog 组件的 Title 属性进行设置来实现的，具体代码如下所示。

```
openFileDialog1.Title = "打开 WINDOWS 文件夹";
```

在图 16.27 中的打开文件对话框比前面例子中的对话框多了一个“帮助”按钮，以及一个文本标签为“以只读方式打开”的单选框。这是通过设置 OpenFileDialog 组件的 ShowReadOnly 属性和 ShowHelp 属性来实现的。具体代码如下所示。



图 16.27 “打开”文件对话框


```
private void button7_Click(object sender, EventArgs e)
{
    //指示在“打开”文件对话框中显示“以只读方式打开”复选框
    openFileDialog1.ShowReadOnly=true;
    //指示在“打开”文件对话框中显示“帮助”按钮
    openFileDialog1.ShowHelp = true;
    DialogResult dr = this.openFileDialog1.ShowDialog();
    if (dr == DialogResult.OK)
    {
        //启动“记事本”打开用户选择的文件
        System.Diagnostics.Process.Start("notepad",
            openFileDialog1.FileName);
    }
}
```

程序编译运行后的结果如图 16.27 所示，程序中使用了如下代码对 ShowReadOnly 属性和 ShowHelp 属性进行赋值：

```
openFileDialog1.ShowReadOnly=true;
openFileDialog1.ShowHelp = true;
```

在上面的程序中，当用户单击“确定”按钮关闭对话框后，程序中使用了 System.Diagnostics.Process.Start() 方法来打开“记事本”，并使用记事本读取用户选择以只读方式打开的文件，具体代码如下所示。

```
System.Diagnostics.Process.Start("notepad", openFileDialog1.FileName);
```

说明: System.Diagnostics.Process.Start()方法可以用来启动应用进程。

程序中使用 FileName 属性直接指定了要打开文件的名字。

16.2.7 检查指定的文件是否存在

在程序运行阶段,用户不是在列表视图中选择要打开的文件,而是通过在文件名文本框中直接输入希望打开的文件名称来直接打开文件,将程序 OpenFileDialog 组件的 CheckFileExists 属性设置为 true,来检查要打开的文件是否存在。与 CheckFileExists 属性类似的还有 CheckPathExists 属性,用于检查指定的路径是否存在,示例代码如下:

```
private void button8_Click(object sender, EventArgs e)
{
    //检查用户指示的路径是否存在
    openFileDialog1.CheckPathExists=true;
    //检查用户指示的文件是否存在
    openFileDialog1.CheckFileExists = true;
    openFileDialog1.FileName = "csharp";
    DialogResult dr = this.openFileDialog1.ShowDialog();
    if (dr == DialogResult.OK)
    {
        //启动“记事本”打开用户选择的文件
        System.Diagnostics.Process.Start("notepad",openFileDialog1.FileName);
    }
}
```

在上面的代码中,使用 FileName 属性指定了要打开的文本文件,但是这个文件并不存在,这时程序将弹出一个如图 16.28 所示的消息对话框,提示文件并不存在。可以单击“确定”按钮关闭消息对话框。

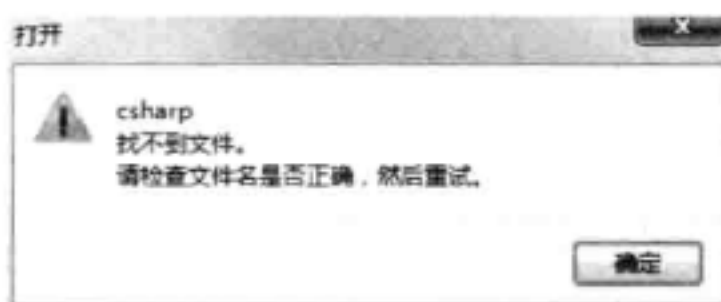



图 16.28 运行结果

注意: 这两个对话框虽然都是用来提示所要打开的文件是不存在的,但是引发的机制却不相同,第一个对话框是由 OpenFileDialog 组件所产生的,而第二个对话框则是由 System.Diagnostics.Process.Start()方法产生的。

如果将上面的示例程序改为如下代码:

```
private void button8_Click(object sender, EventArgs e)
{
    //检查用户指示的路径是否存在
    openFileDialog1.CheckPathExists=true;
    //不检查用户指示的文件是否存在
    openFileDialog1.CheckFileExists = false;
    openFileDialog1.FileName = "csharp";
    DialogResult dr = this.openFileDialog1.ShowDialog();
```



```

if (dr == DialogResult.OK)
{
    //启动“记事本”打开用户选择的文件
    System.Diagnostics.Process.Start("notepad", openFileDialog1.FileName);
}
}

```

那么编译并运行程序后，OpenFileDialog 组件就不会产生警告消息，程序只会弹出如图 16.29 所示的对话框。

若将上段代码中的 System.Diagnostics.Process.Start() 进行替换，变为如下所示的代码，那么程序将不会弹出警告对话框，运行结果如图 16.30 所示。



图 16.29 运行结果



图 16.30 运行结果

```

private void button8_Click(object sender, EventArgs e)
{
    openFileDialog1.CheckPathExists=true;
    openFileDialog1.CheckFileExists = false;
    openFileDialog1.FileName = "csharp";
    DialogResult dr = this.openFileDialog1.ShowDialog();
    if (dr == DialogResult.OK)
    {
        //弹出消息对话框，显示用户选择的文件名
        MessageBox.Show(openFileDialog1.FileName);
    }
}

```

16.2.8 使用 SaveFileDialog 组件

“另存为”对话框是 Windows 系统中用于保存文件的通用对话框。可以通过实例化 System.Windows.Forms.SaveFileDialog 类获得一个保存文件的通用对话框。如图 16.31 所示，就是一个通用的“另存为”对话框。



图 16.31 “另存为”对话框

SaveFileDialog 类的命名空间为 System.Windows.Forms，它在 Visual Studio 工具箱中的图示如图 16.32 所示，将 SaveFileDialog 组件拖曳到窗体后，可看到 SaveFileDialog 组件放置在窗体下方的组件托盘区中，如图 16.33 所示。下一步，可通过对 SaveFileDialog 的属性进行设置，使其达到应用程序的具体需求。



图 16.32 SaveFileDialog 组件



图 16.33 SaveFileDialog 组件

16.2.9 SaveFileDialog 组件编程示例

通常，在 Windows 应用程序中，都是在单击某个按钮或者菜单上的相应选项后才会打开“另存为”对话框。在下面的示例代码中，将生成一个窗体，窗体的具体外观如图 16.34 所示，单击窗体中的“保存”按钮，将打开一个保存文件对话框，并通过此对话框将窗体中文本框的内容保存为一个名为 test.txt 的文本文件。



图 16.34 示例程序窗体

```
private System.Windows.Forms.SaveFileDialog saveFileDialog1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.Button button2;
public Form1()
{
    InitializeComponent();
    this.saveFileDialog1 = new System.Windows.Forms.SaveFileDialog();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.button1 = new System.Windows.Forms.Button();
    this.button2 = new System.Windows.Forms.Button();
    //
    //textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(13, 13);
    this.textBox1.Multiline = true;
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(180, 128);
    this.textBox1.TabIndex = 0;
    this.textBox1.Text = "通常说来，在 Windows 应用程序中，都是在单击某个按钮或者菜单上的相应选项后会打开“打开文件对话框”。在下面的按钮 click 处理事件中，将打开一
```



```

    个文件对话框，并将所选择的文件的名称使用消息对话框显示出来。";
    //
    //button1
    //
    this.button1.Location = new System.Drawing.Point(13, 154);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 1;
    this.button1.Text = "保存";
    this.button1.Click += new System.EventHandler(this.button1_Click);
    //
    //button2
    //
    this.button2.Location = new System.Drawing.Point(118, 154);
    this.button2.Name = "button2";
    this.button2.Size = new System.Drawing.Size(75, 23);
    this.button2.TabIndex = 2;
    this.button2.Text = "退出";

    this.Controls.Add(this.button2);
    this.Controls.Add(this.button1);
    this.Controls.Add(this.textBox1);
}

private void button1_Click(object sender, EventArgs e)
{
    string fileContent = this.textBox1.Text;
    saveFileDialog1.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
    saveFileDialog1.FilterIndex = 1;

    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        string filepath = saveFileDialog1.FileName;
        System.IO.StreamWriter streamWriter =
            new System.IO.StreamWriter(filepath);
        streamWriter.Write(this.textBox1.Text);
        streamWriter.Flush();
        streamWriter.Close();
    }
}

private void button2_Click(object sender, EventArgs e)
{
    this.Close();
}
}

```

对程序进行编译并运行，在图 16.34 所示窗体中单击“保存”按钮，将显示的文本内容保存为 test.txt，打开文件后可看到其内容如图 16.35 所示。

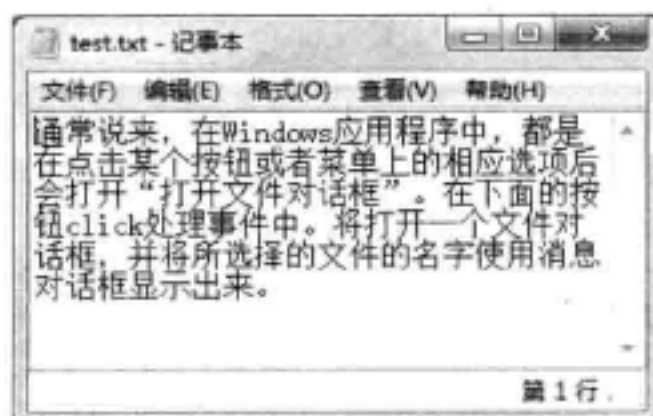


图 16.35 保存文件

16.2.10 另存文件时的覆盖与新建

在使用 `SaveFileDialog` 组件保存文件时，如果要保存文件的文件名与程序中已有文件的文件名发生冲突，保存文件对话框将弹出如图 16.36 所示的对话框，询问用户应该执行怎样的操作。开发人员可通过设置 `SaveFileDialog` 组件中的 `OverwritePrompt` 属性来指示当这种情况发生时，程序是应该弹出对话框进行询问还是直接覆盖原有文件中的内容。

在使用 `SaveFileDialog` 组件保存文件时，如果指定要保存文件的文件名在应用程序中并不存在，那么程序将自动创建新的文件用以保存内容。开发人员可通过设置 `SaveFileDialog` 组件中的 `CreatePrompt` 属性来指示当这种情况发生时，程序是应该直接创建新的文件还是弹出对话框进行询问。如果程序弹出对话框进行询问，那么询问的对话框如图 16.37 所示。



图 16.36 消息对话框



图 16.37 消息对话框

技巧： `OverwritePrompt` 和 `CreatePrompt` 属性最好不要同时设置为 `true`。

在下面的代码中，开发人员实例化了一个新的 `SaveFileDialog` 组件，并在用户单击窗体上的按钮时，将弹出保存文件对话框。在使用此对话框保存文件时，如果用户指定的文件名并不存在，那么对话框将弹出消息对话框，询问是否创建新的文件以保存内容；如果用户指定的文件名存在，那么对话框将直接覆盖原有的文件，并不进行询问。

```
private void button1_Click(object sender, EventArgs e)
{
    SaveFileDialog saveFileDialog1 = new SaveFileDialog();
    //程序直接创建新的文件
    saveFileDialog1.CreatePrompt = true;
    //当保存文件名已经存在时，直接覆盖原有文件中的内容
    saveFileDialog1.OverwritePrompt = false;
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        string filepath = saveFileDialog1.FileName;
        //向指定的文件写入文本框中的内容
        System.IO.StreamWriter streamWriter = new System.IO.StreamWriter(
            filepath);
        streamWriter.Write(this.textBox1.Text);
        streamWriter.Flush();
        streamWriter.Close();
    }
}
```

16.2.11 使用 `FolderBrowserDialog` 组件

“浏览文件夹”对话框是 Windows 系统中用于浏览文件夹的通用对话框。可以通过实

例化 `System.Windows.Forms.FolderBrowserDialog` 类获得一个浏览文件夹通用对话框。如图 16.38 所示，就是一个通用的“浏览文件夹”对话框。



图 16.38 “浏览文件夹”对话框

`FolderBrowserDialog` 类的命名空间为 `System.Windows.Forms`，它在 Visual Studio 工具箱中的图示如图 16.39 所示，将 `FolderBrowserDialog` 组件拖曳到窗体后，可看到 `FolderBrowserDialog` 组件放置在窗体下方的组件托盘区中，如图 16.40 所示。下一步，可通过对 `FolderBrowserDialog` 的属性进行设置，使其达到应用程序的具体需求。



图 16.39 FolderBrowserDialog 组件

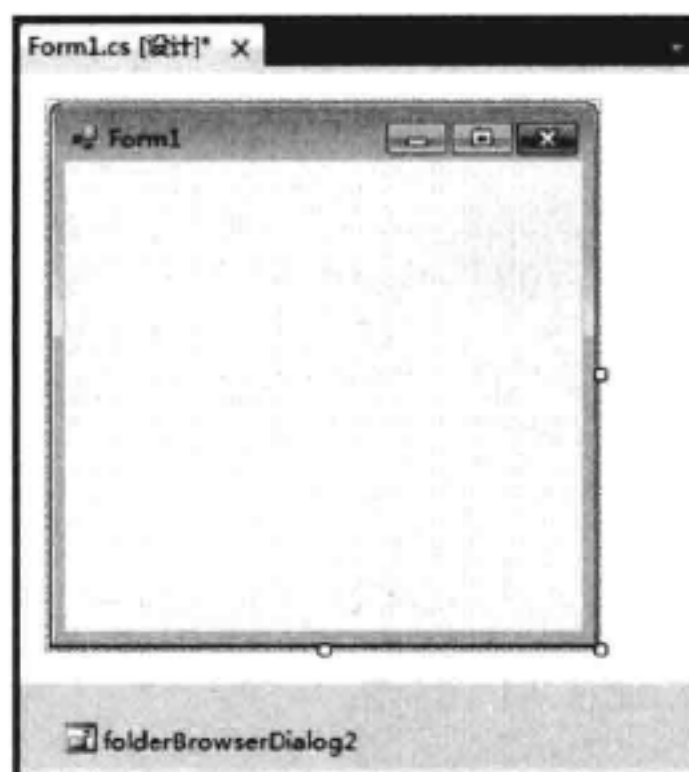


图 16.40 FolderBrowserDialog 组件

16.2.12 FolderBrowserDialog 组件编程示例

在下面的示例代码中，将生成一个窗体，窗体的具体外观如图 16.41 所示。单击窗体中的“查找”按钮，将打开一个“浏览文件夹”对话框，在此对话框中选定了文件夹后，单击“确定”按钮，文件夹路径将随后显示在窗体上的 `TextBox` 控件中。

说明：在 Windows 应用程序中，都是通过单击某个按钮或者菜单上的相应选项后会打开“浏览文件夹”对话框。

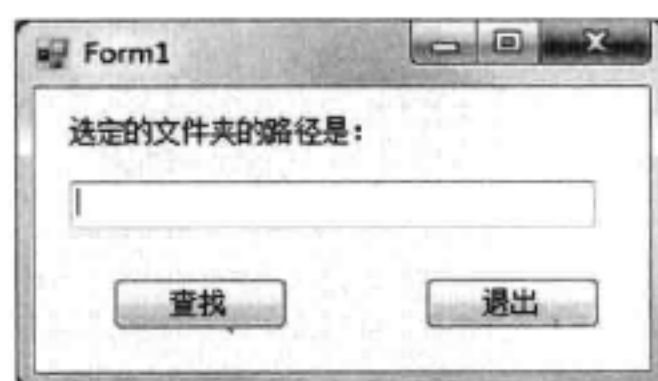


图 16.41 示例程序窗体

```

//定义窗体中的控件
private System.Windows.Forms.FolderBrowserDialog folderBrowserDialog1;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.Button button2;
public Form1()
{
    //对控件赋值
    InitializeComponent();
    this.folderBrowserDialog1 =
        new System.Windows.Forms.FolderBrowserDialog();
    this.label1 = new System.Windows.Forms.Label();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.button1 = new System.Windows.Forms.Button();
    this.button2 = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    //label1
    //
    this.label1.AutoSize = true;
    this.label1.Location = new System.Drawing.Point(13, 13);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(137, 12);
    this.label1.TabIndex = 0;
    this.label1.Text = "选定的文件夹的路径是: ";
    //
    //textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(15, 41);
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(224, 21);
    this.textBox1.TabIndex = 1;
    //
    //button1
    //
    this.button1.Location = new System.Drawing.Point(33, 83);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 2;
    this.button1.Text = "查找";
    this.button1.UseVisualStyleBackColor = true;
    this.button1.Click += new System.EventHandler(this.button1_Click);
    //
    //button2
    //
    this.button2.Location = new System.Drawing.Point(166, 83);
    this.button2.Name = "button2";
    this.button2.Size = new System.Drawing.Size(75, 23);
    this.button2.TabIndex = 3;
    this.button2.Text = "退出";
    this.button2.UseVisualStyleBackColor = true;
    this.button2.Click += new System.EventHandler(this.button2_Click);
    //向窗体添加控件
    this.Controls.Add(this.button2);
    this.Controls.Add(this.button1);
    this.Controls.Add(this.textBox1);
    this.Controls.Add(this.label1);
}
//单击按钮一

```



```

private void button1_Click(object sender, EventArgs e)
{
    //显示新建文件夹按钮
    folderBrowserDialog1.ShowNewFolderButton = true;
    //设置初始文件夹
    folderBrowserDialog1.RootFolder = Environment.SpecialFolder.Desktop;
    //对保存文件对话框进行描述
    folderBrowserDialog1.Description = "选择文件夹";
    if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = folderBrowserDialog1.SelectedPath;
    }
}
//单击按钮二
private void button2_Click(object sender, EventArgs e)
{
    //关闭文件夹
    this.Close();
}

```

对以上程序进行编译并执行，单击运行窗体的“查找”按钮，将弹出如图 16.42 所示的“浏览文件夹”对话框。使用下面的代码对 FolderBrowserDialog 组件进行设置。



图 16.42 “浏览文件夹”对话框

```

//显示新建文件夹按钮
folderBrowserDialog1.ShowNewFolderButton = true;
//设置初始文件夹
folderBrowserDialog1.RootFolder = Environment.SpecialFolder.Desktop;
//对保存文件对话框进行描述
folderBrowserDialog1.Description = "选择文件夹";

```

ShowNewFolderButton 属性用来指示在浏览文件夹对话框中是否显示“新建文件夹”按钮。Description 属性是一个字符串类型的属性，用来指示在对话框上方所显示的描述性语言。RootFolder 属性则表示开始浏览的根文件夹。

SelectedPath 属性是一个包含选定文件夹路径的字符串，它代表了目前所选文件夹的路径。下面的编程语句使用户所选的文件夹的路径可以在窗体的文本框中显示。

```

textBox1.Text = folderBrowserDialog1.SelectedPath;

```

技巧：在编程过程中，开发人员可以通过将 SelectedPath 设置为最初选定的 RootFolder 子文件夹的绝对路径，将 RootFolder 设置为开始浏览的位置。

16.3 字体选择对话框

本节将对字体选择对话框进行详细的介绍。

16.3.1 使用 FontDialog 组件

“字体”选择对话框是 Windows 系统中用于选择字体的通用对话框。可以通过实例化 `System.Windows.Forms.FontDialog` 类获得一个字体选择对话框。如图 16.43 所示就是一个通用的“字体”选择对话框。在此对话框中，显示了字体、字体样式和字体大小的列表框，以及删除线和下划线等效果的复选框、字体外观的示例和字体集的下拉列表框。

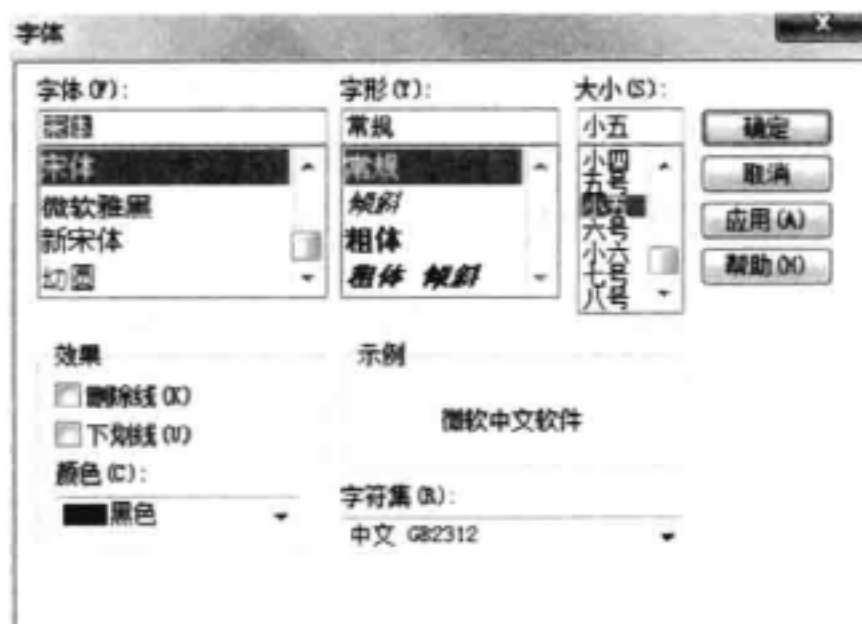


图 16.43 “字体”选择对话框

`FontDialog` 类的命名空间为 `System.Windows.Forms`，它在 Visual Studio 工具箱中的图示如图 16.44 所示，将 `FontDialog` 组件拖曳到窗体后，可看到 `FontDialog` 组件放置在窗体下方的组件托盘区中，如图 16.45 所示。下一步，可通过对 `FontDialog` 的属性进行设置，使其达到应用程序的具体需求。

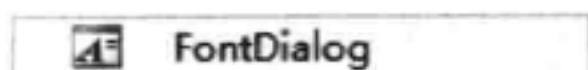


图 16.44 FontDialog 组件

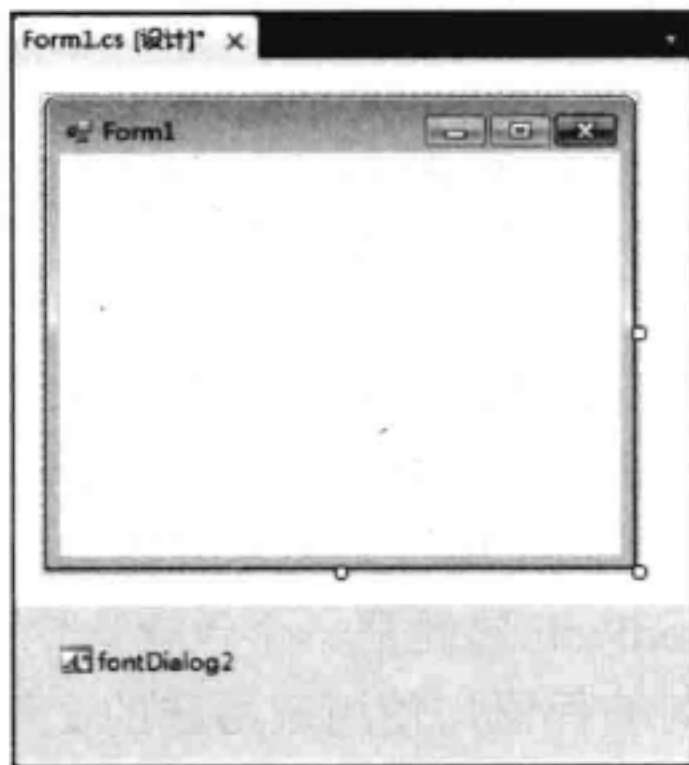



图 16.45 FontDialog 组件

在下面的代码中，`Button` 控件的 `Click` 事件处理程序打开一个 `FontDialog` 组件，如图 16.43 所示。当用户选定字体并单击“确定”按钮时，窗体上的 `TextBox` 控件的 `Font` 属

性被设置为选定的字体。

说明：在 Windows 应用程序中，都是通过单击某个按钮或者菜单上的相应选项后会打开“字体”选择对话框。

```
private void button1_Click(object sender, EventArgs e)
{
    if (fontDialog1.ShowDialog() == DialogResult.OK)
    {
        textBox1.Font = fontDialog1.Font;
    }
}
```

16.3.2 FontDialog 组件编程示例

除了前面介绍的字体选择对话框的基本属性外，字体选择对话框还提供了其他可配置的外观属性。在下面的代码中对这些非属性信息进行设置，使字体选择对话框不仅能选择适当的字体，还能选择适当的字体颜色，具体代码如下所示。

```
private System.Windows.Forms.FontDialog fontDialog1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.TextBox textBox1;
public Form1()
{
    //初始化窗体控件
    InitializeComponent();
    this.fontDialog1 = new System.Windows.Forms.FontDialog();
    this.button1 = new System.Windows.Forms.Button();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    //button1
    //
    this.button1.Location = new System.Drawing.Point(40, 62);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "字体设置";
    this.button1.UseVisualStyleBackColor = true;
    this.button1.Click += new System.EventHandler(this.button1_Click);
    //
    //textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(12, 12);
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(123, 21);
    this.textBox1.TabIndex = 1;
    this.textBox1.Text = "文本设置";
    this.Controls.Add(this.textBox1);
    this.Controls.Add(this.button1);
}
private void button1_Click(object sender, EventArgs e)
{
    //在字体对话框中显示颜色选择
    fontDialog1.ShowColor = true;
    //在字体对话框中显示应用按钮
}
```

```

fontDialog1.ShowDialog() == DialogResult.OK)
{
    textBox1.Font = fontDialog1.Font;
    textBox1.ForeColor = fontDialog1.Color;
}
}

```

编译并运行上面的程序，得到的程序运行代码如图 16.46 所示，单击“字体设置”按钮，将弹出如图 16.47 所示的“字体”选择对话框。在这个对话框中，不仅显示了字体、字体样式和字体大小的列表框，而且还显示了字体颜色下拉列表、应用按钮和帮助按钮。

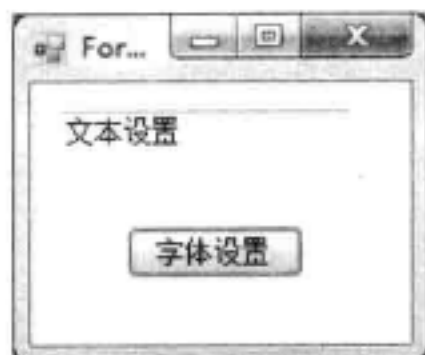


图 16.46 示例程序

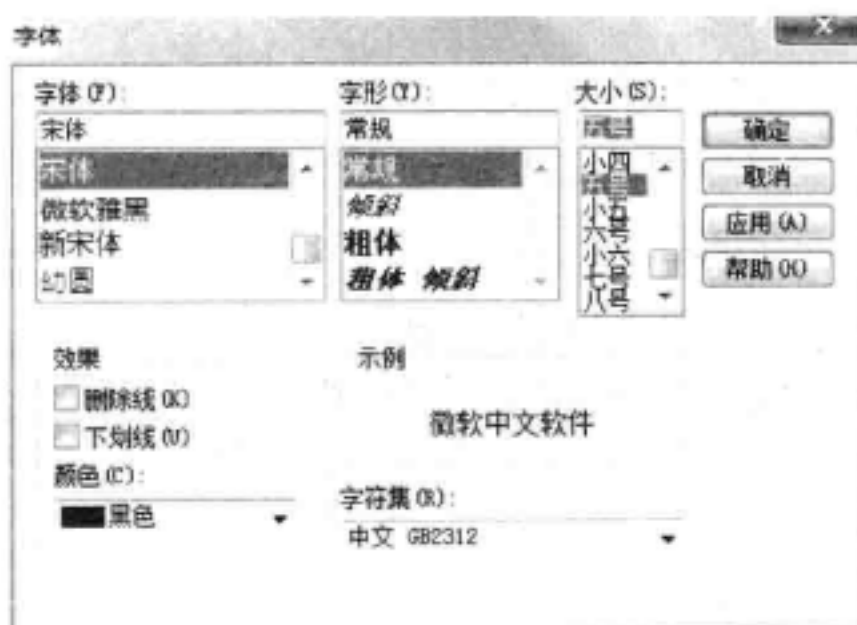


图 16.47 “字体”选择对话框

在上面的示例中，程序通过对 FontDialog 组件的 ShowApply 属性进行设置使对话框中显示了“应用”按钮；通过对 ShowHelp 属性进行设置使对话框中显示了“帮助”按钮；通过对 ShowEffects 属性进行设置使对话框中显示了允许用户指定删除线、下划线和文本颜色选项的控件；通过对 ShowColor 属性进行设置使对话框中显示了颜色选择下拉列表框。具体代码如下所示。

```

//在“字体”对话框中显示颜色选择
fontDialog1.ShowColor = true;
//在“字体”对话框中显示应用按钮
fontDialog1.ShowApply = true;
//在“字体”对话框中显示效果
fontDialog1.ShowEffects = true;
//在“字体”对话框中显示帮助按钮
fontDialog1.ShowHelp = true;

```

在对所选择文本的字体和颜色进行了设置后，程序使用 FontDialog 组件的 Font 属性和 Color 属性来获得用户设置的字体和颜色，并将它赋给文本框中的文字，具体代码如下所示。

```

textBox1.Font = fontDialog1.Font;
textBox1.ForeColor = fontDialog1.Color;

```

如果用户单击了“应用”按钮，将引发 FontDialog 组件的 Apply 事件。在下面的实例程序中，演示了程序如何处理 Apply 事件，具体代码如下：

```

private void button1_Click(System.Object sender, System.EventArgs e)
{

```



```

fontDialog1.FontMustExist = true;
fontDialog1.MaxSize = 32;
fontDialog1.MinSize = 18;
fontDialog1.ShowApply = true;
fontDialog1.ShowEffects = false;
System.Drawing.Font font = this.textBox1.Font;
fontDialog1.Apply += new System.EventHandler(FontDialog1_Apply);
if (fontDialog1.ShowDialog() == DialogResult.OK)
{
    fontDialog1_Apply(this.button1, new System.EventArgs());
}
else
{
    if (fontDialog1.ShowDialog() == DialogResult.Cancel)
    {
        this.textBox1.Font = font;
    }
}
}
private void fontDialog1_Apply(object sender, System.EventArgs e)
{
    this.textBox1.Font = fontDialog1.Font;
}

```

在上面的代码中，通过设置 FontDialog 组件的 MaxSize 属性和 MinSize 属性来设置在字体选择对话框中用户可选择的最大字体磅值和最小字体磅值。

16.4 颜色选择对话框

本节将对颜色选择对话框进行详细的介绍。

16.4.1 使用 ColorDialog 组件

“颜色”选择对话框是 Windows 系统中用于选择颜色的通用对话框。可以通过实例化 System.Windows.Forms.ColorDialog 类获得一个颜色选择对话框。如图 16.48 和图 16.49 所示就是一个通用的“颜色”选择对话框。在此对话框中，用户可以从调色板选择颜色，以及将自定义颜色添加到该调色板中。在颜色选择对话框中一共包括两部分：一部分显示基本颜色，另一部分允许用户定义自定义颜色。

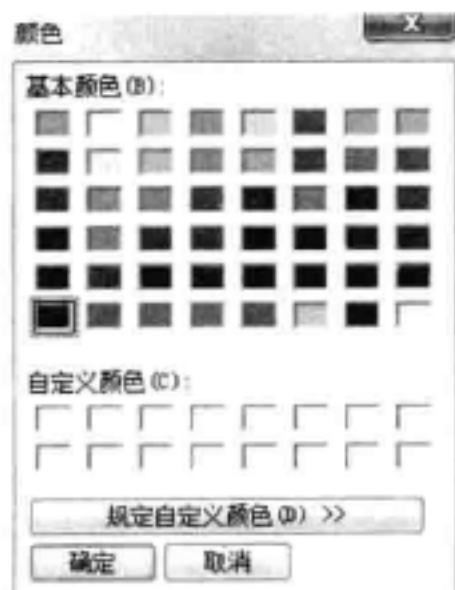


图 16.48 “颜色”选择对话框 (1)

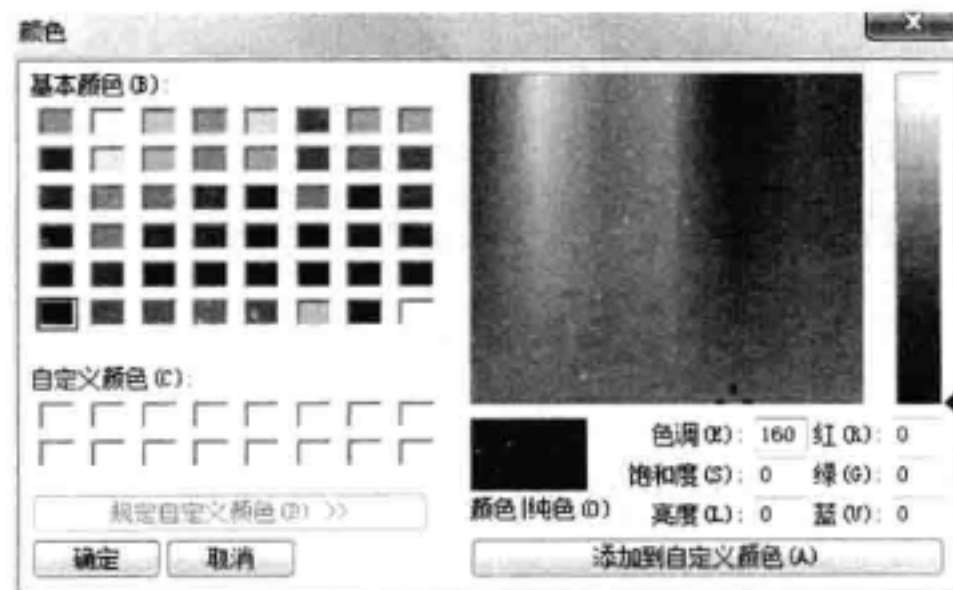


图 16.49 “颜色”选择对话框 (2)

ColorDialog 类的命名空间为 System.Windows.Forms，它在 Visual Studio 工具箱中的图示如图 16.50 所示，将 ColorDialog 组件拖曳到窗体后，可看到 ColorDialog 组件放置在窗体下方的组件托盘区中，如图 16.51 所示。下一步，可通过对 ColorDialog 的属性进行设置，使其达到应用程序的具体需求。

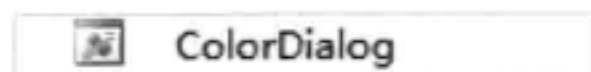


图 16.50 ColorDialog 组件

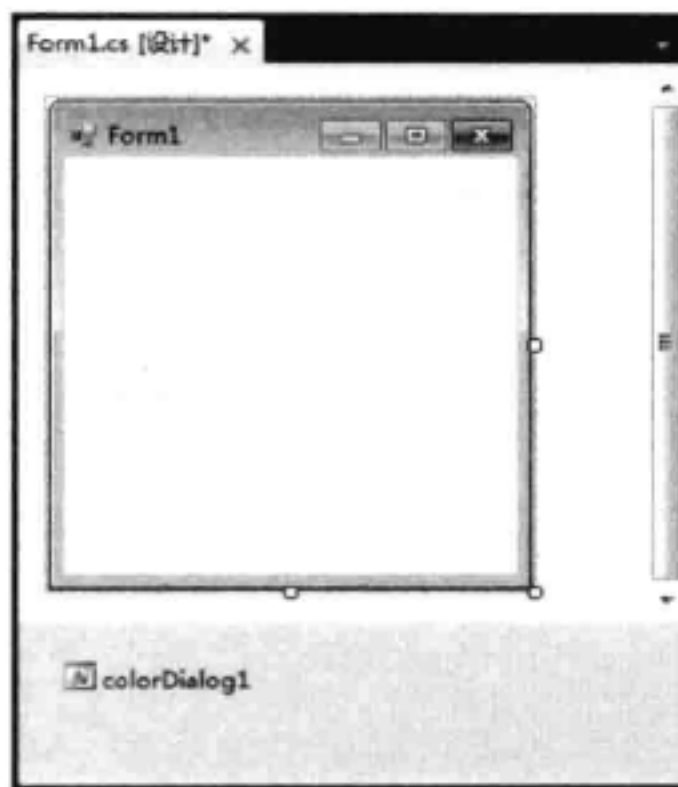


图 16.51 ColorDialog 组件

说明：开发人员可以通过使用 ColorDialog 组件的属性来配置颜色选择对话框的外观。在使用此对话框选择颜色时，用户在对话框中选择的颜色可以通过 ColorDialog 组件的 Color 属性来获得。

- ❑ 如果将 ColorDialog 组件的 AllowFullOpen 属性设置为 false，那么将禁止使用“定义自定义颜色”按钮，用户只能选择调色板中的预定义颜色。
- ❑ 如果将 ColorDialog 组件的 SolidColorOnly 属性设置为 true，则用户不能选择仿色（仿造颜色，即用较少的颜色来表达较丰富的色彩过渡），只有纯色（即组成可见光谱的单色）可供选择。
- ❑ 如果将 ColorDialog 组件的 AnyColor 属性设置为 true，那么颜色选择对话框的基本颜色集内将显示所有可用的颜色。
- ❑ 如果对话框允许用户定义自定义颜色的部分被打开，那么 FullOpen 属性为 true；程序可以通过捕获 FullOpen 属性的属性值来获取选择颜色对话框当前的状态。
- ❑ 如果将 ShowHelp 属性设置为 true，则会在对话框上显示“帮助”按钮。

下面的代码通过将 AllowFullOpen、AnyColor、SolidColorOnly 和 ShowHelp 属性设置为期望值来配置颜色对话框的外观。

```
colorDialog1.AllowFullOpen = true;
colorDialog1.AnyColor = true;
colorDialog1.SolidColorOnly = false;
colorDialog1.ShowHelp = true;
```

设置后的颜色选择对话框如图 16.52 和图 16.53 所示。

16.4.2 ColorDialog 组件编程示例

在下面的程序中，将在 Button 控件的 Click 事件处理程序打开一个 ColorDialog 组件。

当用户选定颜色并单击“确定”按钮后，Button 控件的背景色将设置为选定的颜色。

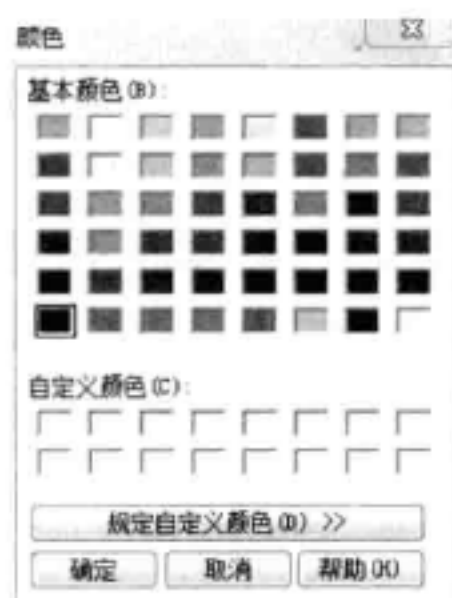


图 16.52 “颜色”选择对话框 (3)

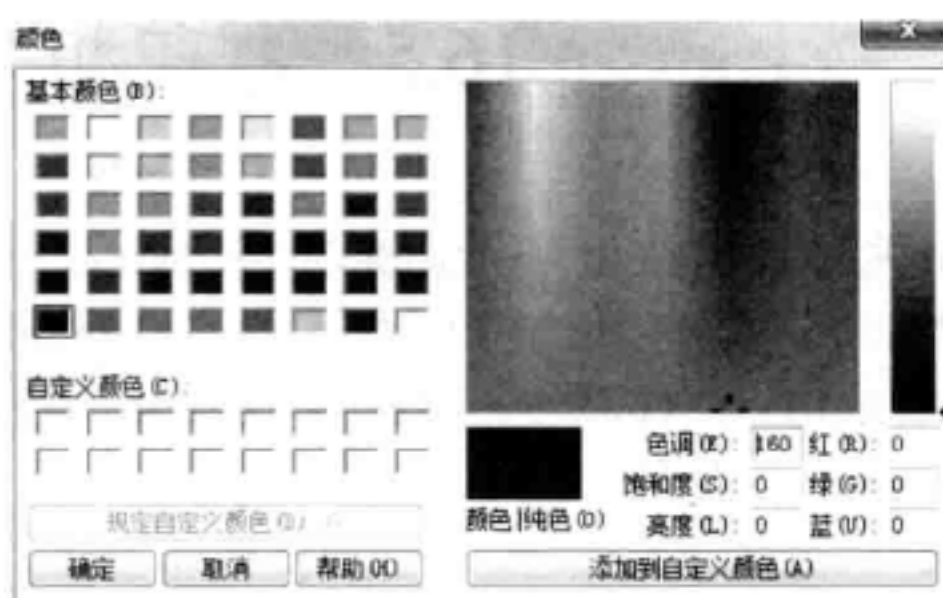



图 16.53 “颜色”选择对话框 (4)

```
private void button1_Click(object sender, System.EventArgs e)
{
    if(colorDialog1.ShowDialog() == DialogResult.OK)
    {
        //将在“颜色”选择对话框中选择的颜色设为按钮的背景色
        button1.BackColor = colorDialog1.Color;
    }
}
```

下面的代码设置了一个不允许用户设置自定义颜色，但允许显示完整的基本颜色集的颜色选择对话框。这段代码还说明了如何设置 ShowHelp 属性，以及如何处理对话框的 HelpRequest 事件。

```
private void InitializeColorDialog()
{
    this.colorDialog1 = new System.Windows.Forms.ColorDialog();
    //并不完全展开“颜色”选择对话框
    this.colorDialog1.AllowFullOpen = false;
    //“颜色”选择对话框的基本颜色集内将显示所有可用的颜色
    this.colorDialog1.AnyColor = true;
    //用户可以选择仿色
    this.colorDialog1.SolidColorOnly = false;
    //显示“帮助”按钮
    this.colorDialog1.ShowHelp = true;
    this.colorDialog1.HelpRequest +=
        new System.EventHandler(colorDialog1_HelpRequest);
}
private void button1_Click(System.Object sender, System.EventArgs e)
{
    if (colorDialog1.ShowDialog().Equals(DialogResult.OK))
    {
        textBox1.BackColor = colorDialog1.Color;
    }
}
//在用户单击“帮助”按钮时被引发
private void colorDialog1_HelpRequest(object sender, System.EventArgs e)
{
    MessageBox.Show("请选定一种颜色 "+ "它将重新设置文本框的背景色 .");
}
```

 注意：HelpRequest 事件是向应用程序添加帮助系统时的常用事件。

16.5 打印相关对话框

打印相关对话框主要是指用来完成 Windows 应用程序的打印功能的相关对话框，在 .NET 中，主要有用于页面设置的 `PageSetupDialog` 组件、用于打印预览的 `PrintPreviewDialog` 控件和 `PrintPreviewControl` 控件、用于打印设置的 `PrintDialog` 组件和 `PrintDocument` 组件几种。

16.5.1 使用 `PrintDocument` 组件

`PrintDocument` 组件用于设置描述打印内容的属性，并且可以用于在基于 Windows 的应用程序中打印文档。可以将它与 .NET 中的其他打印控件一起使用。其他打印控件可以通过 Windows 通用打印对话框使用户来对 `PrintDocument` 组件的各个属性进行设置，从而控制文档打印的各个方面。

`PrintDocument` 类的命名空间为 `System.Drawing.Printing`，它在 Visual Studio 工具箱中的图示如图 16.54 所示，将 `PrintDocument` 组件拖曳到窗体后，可看到 `PrintDocument` 组件放置在窗体下方的组件托盘区中，如图 16.55 所示。下一步，可通过对 `PrintDocument` 的属性进行设置，使其达到应用程序的具体需求。



图 16.54 `PrintDocument` 组件



图 16.55 `PrintDocument` 组件

在使用 `PrintDocument` 组件直接进行文本打印的时候，`BeginPrint` 事件发生在调用 `PrintDocument` 组件的 `Print()` 方法并且在打印开始之前，`EndPrint` 事件发生在文档打印完之后，而打印当前页时，将触发 `PrintPage` 事件。

说明：当打印某个文本文件的时候，可在 `PrintPage` 事件处理程序中添加打印文件的编程逻辑。

16.5.2 `PrintDocument` 组件编程示例

下面的代码将在默认打印机上打印名为 `c:\c-sharp.txt` 的文件。


```

//引用的命名空间
using System;
using System.IO;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;
public class DocumentPrinting : Form
{
    private System.ComponentModel.Container components;
    private System.Windows.Forms.Button button1;
    private Font font;
    private StreamReader stream;
    //构造函数
    public DocumentPrinting() : base()
    {
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e)
    {
        //异常处理
        try
        {
            //读入 c-sharp.txt 文件的内容
            stream = new StreamReader ("c:\\c-sharp.txt");
            //异常处理
            try
            {
                //打印文档控件
                font = new Font("Arial", 10);
                PrintDocument document = new PrintDocument();
                document.PrintPage +=
                    new PrintPageEventHandler(this.documentPrintPage);
                //打印
                document.Print();
            }
            //在打印过程中出现异常，弹出对话框进行提醒
            catch(Exception e)
            {
                MessageBox.Show("打印出现异常");
            }
            //关闭读文件流
            finally
            {
                stream.Close();
            }
        }
        //在打印过程中出现异常，弹出对话框进行提醒
        catch(Exception ex)
        {
            MessageBox.Show("打印出现异常");
        }
    }
    //打印文档事件
    private void document_PrintPage(object sender, PrintPageEventArgs e)
    {
        //初始化变量
        float lineCount = 0;
        float yCoordinate = 0;
        int count = 0;
    }
}


```

```

//文本打印左边距
float left = e.MarginBounds.Left;
//文本打印上边距
float top = e.MarginBounds.Top;
string text = null;
//确定每页的行数
lineCount = e.MarginBounds.Height / font.GetHeight(e.Graphics);
//逐行进行打印
while(count < lineCount && ((text=stream.ReadLine()) != null))
{
    //打印行的纵坐标
    yCoordinate = top + (count * font.GetHeight(e.Graphics));
    //打印这一行
    e.Graphics.DrawString(text, font, Brushes.Black, left, yCoordinate, new StringFormat());
    count++;
}
//如果要打印的文本不为空
if(text != null)
{
    //进行换页
    e.HasMorePages = true;
}
//如果要打印的文本为空
else
{
    //不换页
    e.HasMorePages = false;
}
}
//初始化控件
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.button1 = new System.Windows.Forms.Button();
    this.ClientSize = new System.Drawing.Size(500, 400);
    this.Text = "打印示例";
    button1.ImageAlign = System.Drawing.ContentAlignment.MiddleLeft;
    button1.Location = new System.Drawing.Point(30, 100);
    button1.FlatStyle = System.Windows.Forms.FlatStyle.Flat;
    button1.TabIndex = 0;
    button1.Text = "打印文本";
    button1.Size = new System.Drawing.Size(150, 50);
    button1.Click += new System.EventHandler(button1_Click);
    this.Controls.Add(button1);
}
//程序入口
public static void Main(string[] args)
{
    Application.Run(new PrintingExample());
}
}

```


上例中，在 `document_PrintPage` 事件的处理逻辑中，通过编码实现了文档中的每一行都适合于页宽。

 **注意：** `PrintDocument` 组件的其他打印功能都可以通过 .NET 提供的通用打印对话框得到实现，在此处不再详细介绍，建议读者使用 .NET 提供的组件完成设置。

16.5.3 使用 `PageSetupDialog` 组件

“页面设置”对话框是 Windows 系统中用于设置打印页面的通用对话框，它为用户提供文档的布局、页面大小，以及其他页面布局选择。可以通过实例化 `System.Windows.Forms.PageSetupDialog` 类获得一个页面设置对话框。如图 16.56 就是一个通用的“页面设置”对话框。



图 16.56 “页面设置”对话框

`PageSetupDialog` 类的命名空间为 `System.Windows.Forms`，它在 Visual Studio.NET 的工具箱中的图示如图 16.57 所示，将 `PageSetupDialog` 组件拖曳到窗体后，可看到 `PageSetupDialog` 组件放置在窗体下方的组件托盘区中，如图 16.58 所示。下一步，可通过对 `PageSetupDialog` 的属性进行设置，使其达到应用程序的具体需求。

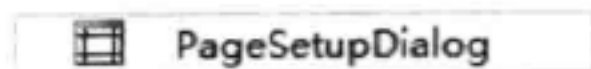


图 16.57 `PageSetupDialog` 组件

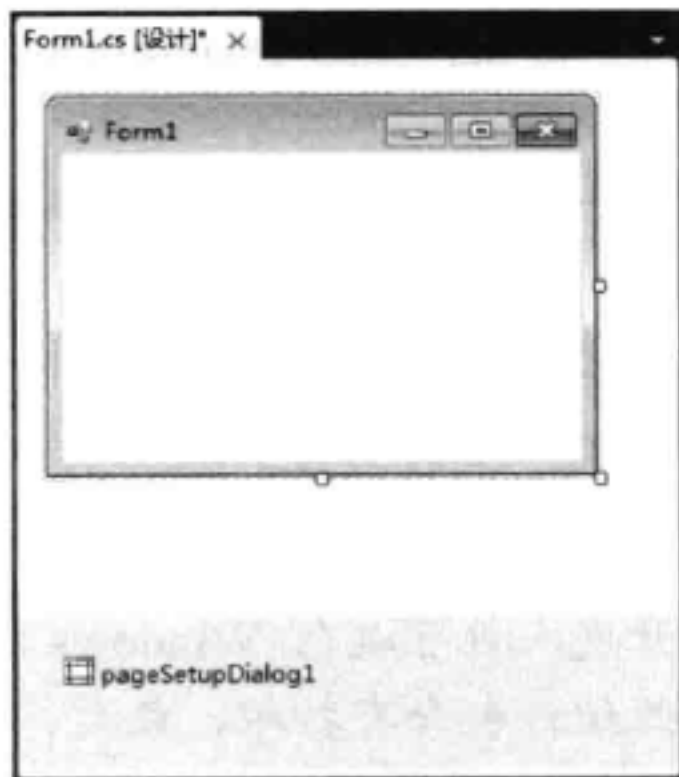


图 16.58 `PageSetupDialog` 组件

在使用 PageSetupDialog 组件时,开发人员必须首先指定一个 PrintDocument 类的实例,这个 PrintDocument 类的实例代表了应用程序要打印的文档。在应用程序中使用 PageSetupDialog 组件,实际上是以可视化的方式提供了让用户自定义修改与即将打印的文档相关联的 PageSettings 类。

16.5.4 PageSetupDialog 组件编程示例


在下面的示例中,在 Button 控件的 Click 事件处理程序中实例化了一个 PageSetupDialog 组件的一个,当用户单击按钮时,将弹出一个“页面设置”对话框,设置代码如下:

```
private void button1_Click(object sender, EventArgs e)
{
    //定义打印的文档
    pageSetupDialog1.Document = this.printDocument1;
    //对 pageSetupDialog1 的属性进行设置
    pageSetupDialog1.Document.DefaultPageSettings.Color = false;
    //“页面设置”对话框将启用对话框的页面方向设置部分
    pageSetupDialog1.AllowOrientation = true;
    //“页面设置”对话框将启用对话框的打印页面边距设置部分
    pageSetupDialog1.AllowMargins = true;
    //“页面设置”对话框将启用对话框的打印纸张设置部分
    pageSetupDialog1.AllowPaper = true;
    //“页面设置”对话框将启用对话框的“打印机”按钮
    pageSetupDialog1.AllowPrinter = true;
    pageSetupDialog1.ShowNetwork = true;
    pageSetupDialog1.ShowDialog();
}
```

在上面的程序中,通过对 pageSetupDialog1 的属性进行定义,设置了“页面设置”对话框的外观。如果将 PageSetupDialog 组件的 AllowOrientation 属性设置为 true,那么“页面设置”对话框将启用对话框的页面方向设置部分。如果将 PageSetupDialog 组件的 AllowMargins 属性设置为 true,那么“页面设置”对话框将启用对话框的打印页面边距设置部分。如果将 PageSetupDialog 组件的 AllowPaper 属性设置为 true,那么“页面设置”对话框将启用对话框的打印纸张设置部分,用于设置打印纸张的大小和来源。如果将 AllowPrinter 组件的 AllowPrinter 属性设置为 true,那么“页面设置”对话框将启用对话框的“打印机”按钮,用以对计算机的“打印机”进行设置。

16.5.5 使用 PrintPreviewDialog 组件

“打印预览”对话框是 Windows 系统中用于显示文档打印后的外观的对话框。可以通过实例化 System.Windows.Forms.PrintPreviewDialog 类获得一个打印预览对话框。如图 16.59 所示就是一个通用的“打印预览”对话框。

 **说明:** 开发人员可以在 Windows 应用程序中,通用此组件使用户在打印前预览文档,此组件包含有打印、放大、显示一页或多页和关闭此对话框等按钮。

PrintPreviewDialog 类的命名空间为 System.Windows.Forms,它在 Visual Studio 工具箱

中的图示如图 16.60 所示，将 PrintPreviewDialog 组件拖曳到窗体后，可看到 PrintPreviewDialog 组件放置在窗体下方的组件托盘区中，如图 16.61 所示。下一步，可通过对 PrintPreviewDialog 的属性进行设置，使其达到应用程序的具体需求。

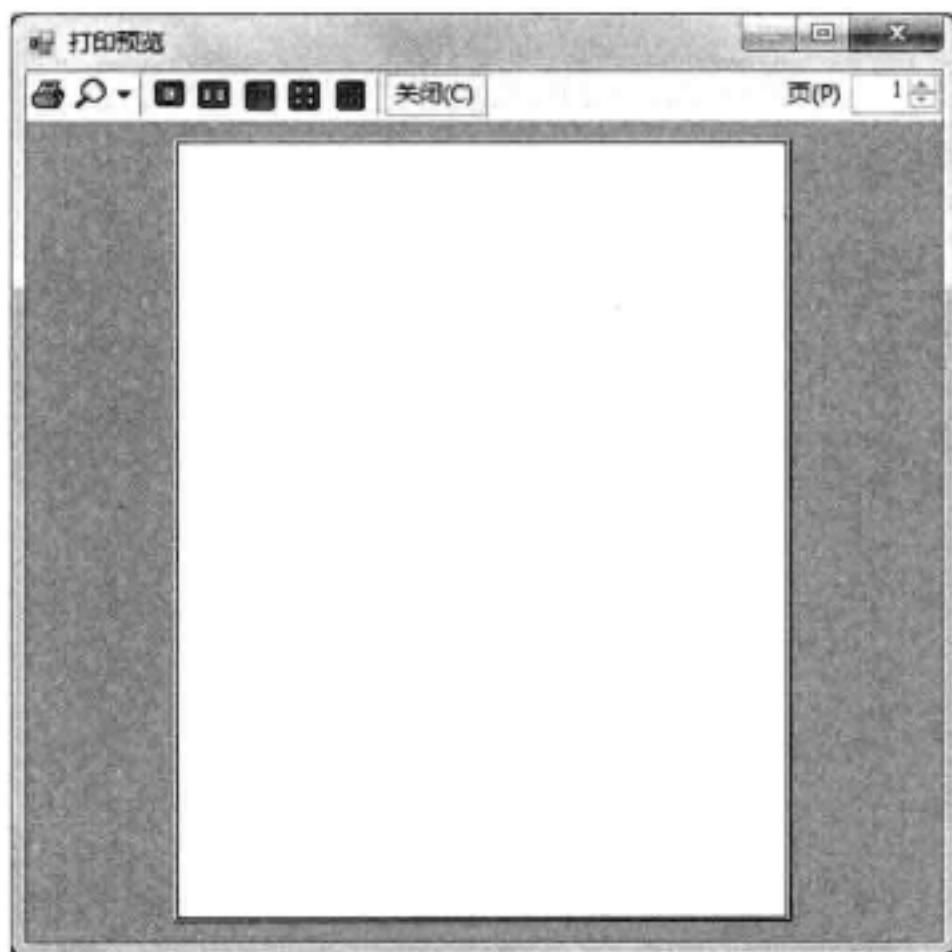


图 16.59 “打印预览”对话框

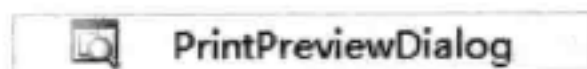


图 16.60 PrintPreviewDialog 组件

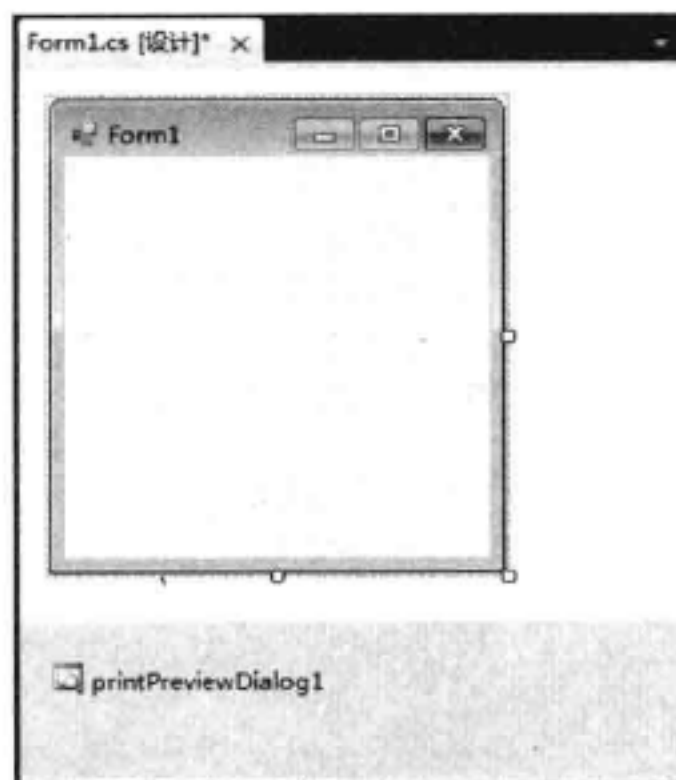


图 16.61 PrintPreviewDialog 组件

在使用 PrintPreviewDialog 组件时，开发人员必须首先指定一个 PrintDocument 类的实例，这个 PrintDocument 类的实例代表了应用程序要打印的文档，并将这个实例赋给 PrintPreviewDialog 的 Document 属性。开发人员还可以通过设置 UseAntiAlias 属性为 true，以消除锯齿使文字显得更齐整平滑，但是同时会使显示更慢。

在下面的代码中，Button 控件的 Click 事件处理程序打开一个打印预览对话框。

```
private void button1_Click(object sender, System.EventArgs e)
{
    printPreviewDialog1.Document = myDocument;
    printPreviewDialog1.ShowDialog();
}
```

在上面的示例中，程序中并没有指定具体要打印的文档，所以单击按钮后，预览窗体

如图 16.59 所示，并没有任何预览内容。

16.5.6 PrintPreviewDialog 组件编程示例

在下面的程序中，将对 PrintPreviewDialog 组件的 Document 属性赋予一个具体的打印文档，使其可以在预览窗体中进行预览，代码如下：

```
//引用的命名空间
using System;
using System.IO;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Printing;
using System.Text;
using System.Windows.Forms;
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        private System.Drawing.Printing.PrintDocument printDocument1;
        private Button button1;
        private StreamReader stream;
        private System.Drawing.Font font;
        private PrintPreviewDialog printPreviewDialog1;
        public Form1()
        {
            InitializeComponent();
            this.printPreviewDialog1 =
                new System.Windows.Forms.PrintPreviewDialog();
            this.printDocument1 =
                new System.Drawing.Printing.PrintDocument();
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            //printPreviewDialog1
            //
            this.printPreviewDialog1.AutoScrollMargin =
                new System.Drawing.Size(0, 0);
            this.printPreviewDialog1.AutoScrollMinSize =
                new System.Drawing.Size(0, 0);
            this.printPreviewDialog1.ClientSize =
                new System.Drawing.Size(400, 300);
            this.printPreviewDialog1.Enabled = true;
            this.printPreviewDialog1.Name = "printPreviewDialog1";
            this.printPreviewDialog1.Visible = false;
            //
            //printDocument1
            //
            this.printDocument1.PrintPage +=
                new PrintPageEventHandler(this.printDocument1_PrintPage);
            //
            //button1
            //
            this.button1.Location = new System.Drawing.Point(31, 43);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
        }
    }
}
```



```

        this.button1.TabIndex = 0;
        this.button1.Text = "button1";
        this.button1.UseVisualStyleBackColor = true;
        this.button1.Click +=
            new System.EventHandler(this.button1_Click);
        this.Controls.Add(this.button1);
    }
    private void button1_Click(object sender, EventArgs e)
    {
        //将“c:\c-sharp.txt”中的内容输入流文件
        stream = new StreamReader("C:\\c-sharp.txt");
        //设置文件的字体
        font = new Font("Arial", 10);
        //设置打印文档的名字
        this.printDocument1.DocumentName = "C:\\c-sharp.txt";
        this.printPreviewDialog1.Document = this.printDocument1;
        this.printPreviewDialog1.ShowDialog();
        //关闭文档
        stream.Close();
    }
    //对文件打印时引发的事件
    private void printDocument1_PrintPage(object sender,
        PrintPageEventArgs e)
    {
        //初始化变量
        float lineCount = 0;
        float yCoordinate = 0;
        int count = 0;
        //文本打印左边距
        float left = e.MarginBounds.Left;
        //文本打印上边距
        float top = e.MarginBounds.Top;
        string text = null;
        //确定每页的行数
        lineCount = e.MarginBounds.Height / font.GetHeight(e.Graphics);
        //逐行进行打印
        while(count < lineCount && ((text=stream.ReadLine()) != null))
        {
            //打印行的纵坐标
            yCoordinate = top + (count * font.GetHeight(e.Graphics));
            //打印这一行
            e.Graphics.DrawString(text, font, Brushes.Black, left,
                yCoordinate, new StringFormat());
            count++;
        }
        //如果要打印的文本不为空, 进行换页
        if(text != null)
        {
            e.HasMorePages = true;
        }
        else
        {
            e.HasMorePages = false;
        }
    }
}

```

```

    }
}

```

编译程序并运行，在结果窗体中单击按钮，将弹出打印预览与如图 16.62 所示的消息对话框，此消息对话框通过文本的变化来显示当前载入的文档页数，载入完毕后，打印预览窗体如图 16.63 所示。

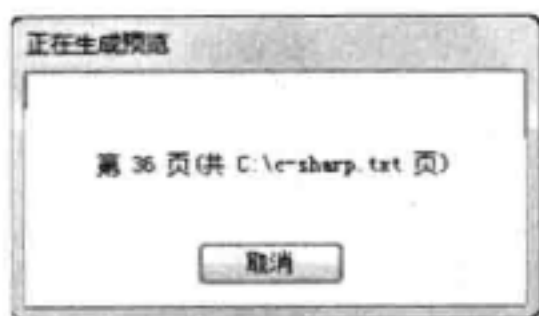


图 16.62 运行结果

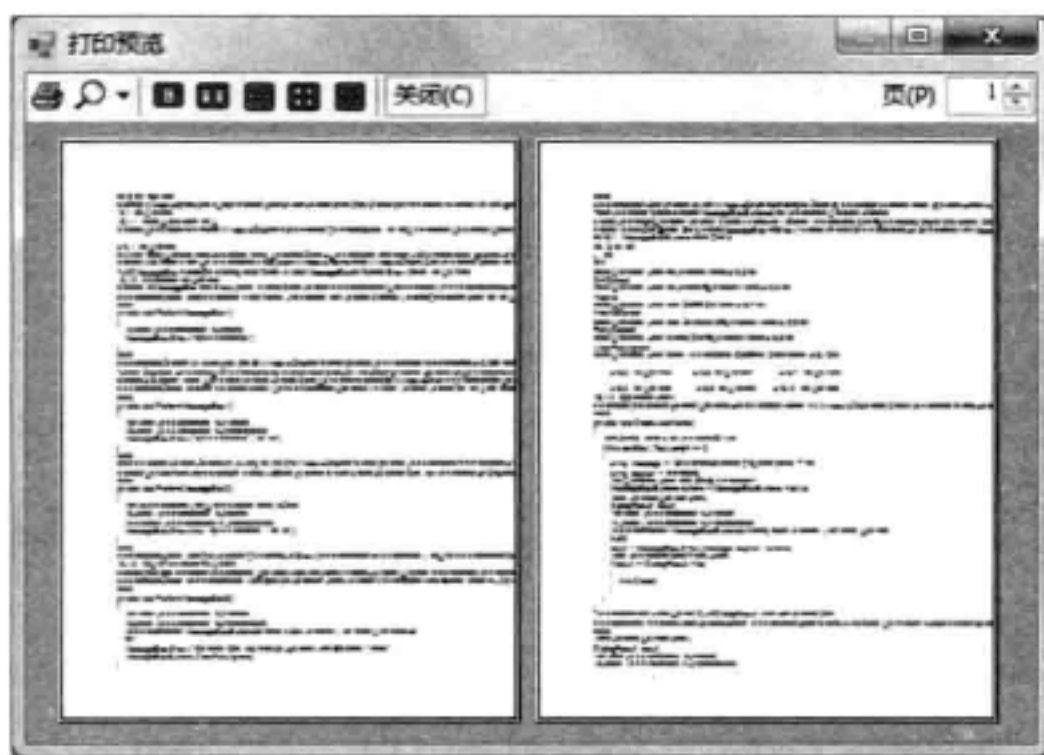


图 16.63 运行结果

16.5.7 使用 PrintDialog 组件

“打印”对话框是在 Windows 应用程序中用于选择打印机、选择要打印的页码，以及确定其他与打印相关设置的对话框。可以通过实例化 System.Windows.Forms.PrintDialog 类获得一个打印对话框。如图 16.64 所示就是一个通用的“打印”对话框。



图 16.64 “打印”对话框

PrintDialog 类的命名空间为 System.Windows.Forms，它在 Visual Studio 工具箱中的图示如图 16.65 所示，将 PrintDialog 组件拖曳到窗体后，可看到 PrintDialog 组件放置在窗体下方的组件托盘区中，如图 16.66 所示。下一步，可通过对 PrintDialog 的属性进行设置，使其达到应用程序的具体需求。



图 16.65 PrintDialog 组件

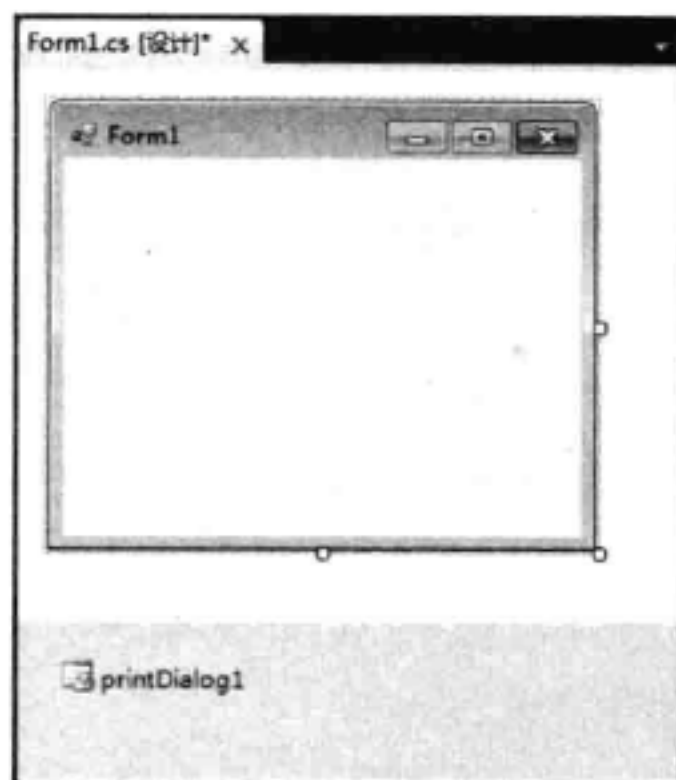


图 16.66 PrintDialog 组件

在下面的代码中，当用户单击窗体按钮时，程序将通过使用 PrintDialog 组件完成了一个简单的打印操作。

```
private System.Drawing.Printing.PrintDocument docToPrint =
    new System.Drawing.Printing.PrintDocument();
private void button1_Click(System.Object sender, System.EventArgs e)
{
    //允许进行多页打印
    printDialog1.AllowSomePages = true;
    //显示帮助按钮
    printDialog1.ShowHelp = true;
    //设置 printDialog1 打印的文档
    printDialog1.Document = docToPrint;
    DialogResult result = printDialog1.ShowDialog();
    if (result==DialogResult.OK)
    {
        docToPrint.Print();
    }
}
private void document1_PrintPage(object sender, System.Drawing.Printing.
PrintPageEventArgs e)
{
    string text = "打印测试";
    System.Drawing.Font printFont= new System.Drawing.Font ("Arial", 10);
    e.Graphics.DrawString(text, printFont, System.Drawing.Brushes.Black,
    10, 10);
}
```

16.6 本章总结

在本节中，主要介绍了 Windows 应用程序的消息对话框和通用对话框，利用这些标准的 Windows 对话框，可以创建基本功能完善且能够立即为用户所熟悉的应用程序。

消息对话框是一个特殊的 Windows 应用程序通用对话框，它主要用来向用户显示提示信息、警告信息等相关信息。

本节中所介绍的通用对话框均是有模式对话框，因此，在显示这些对话框时，它们会

阻止应用程序剩余部分的运行，直到用户做出了设置，关闭了文件夹。

在文件操作对话框的相关组件中，`OpenFileDialog` 组件可以使用户浏览他们的计算机，以及网络中任何计算机上的文件夹，并选择打开一个或多个文件。`SaveFileDialog` 组件可以使用户浏览文件系统并选择要保存的文件。`FolderBrowserDialog` 组件可以使用户浏览、创建并最终选择一个文件夹。

在打印对话框的相关组件中，`PageSetupDialog` 组件可以为用户提供文档的布局、页面大小，以及其他页面布局选择。`PrintPreviewDialog` 组件可以显示文档打印后的外观。`PrintDialog` 组件可以在 Windows 应用程序中选择打印机、选择要打印的页码，以及确定其他与打印相关的设置。

字体选择对话框可以为用户显示字体、字体样式和字体大小的列表框；删除线和下划线等效果的复选框；脚本的下拉列表，以及字体外观的示例。

颜色选择对话框可以使用户从调色板选择颜色，并将自定义颜色添加到该调色板。

16.7 实战练习

1. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在窗体 `Form1` 中添加一个 `TextBox` 控件，再添加一个“浏览”按钮，当用户单击“浏览”按钮时显示一个文件夹浏览窗体，当用户在该窗体中选择文件夹名称后，该名称将显示在 `Form1` 窗体的 `TextBox` 文本框中。

 提示：使用 `FolderBrowserDialog` 组件用于选择文件夹。

2. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在窗体 `Form1` 中添加一个 `TextBox` 控件，再添加一个“颜色”按钮，当用户单击“颜色”按钮时显示一个颜色选择窗体，当用户选择好颜色之后，将 `TextBox` 文本框中文本的颜色设置为选择的颜色。

3. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在窗体 `Form1` 中添加一个显示多行文本的 `TextBox` 控件，再添加一个“字体”按钮，当用户单击“字体”按钮时显示一个字体窗体，通过该字体窗体设置 `TextBox` 控件中的字体。

4. 接第 3 题，在窗体 `Form1` 中再添加一个“保存”按钮，单击该“保存”按钮时将显示一个保存文件对话框，让用户选择保存文件的位置和输入文件名，用来保存 `TextBox` 中的文本。

5. 接第 4 题，在窗体 `Form1` 中再添加一个“打印预览”按钮，当用户单击“打印预览”按钮时将上题中保存的文件在打印预览窗口中显示出来。

第 17 章 其他常用控件


在本书前面几章中介绍了许多控件，本章还将重点介绍一些在 Windows 应用程序中经常使用的其他类控件。如菜单控件、计时器控件、图形控件等。

17.1 计时器组件

Timer（计时器）组件也是一个 WinForm 组件，和其他 WinForm 组件的最大区别是：Timer 组件是不可见的，而其他大部分的组件都是可见的，是可以设计的。当 Timer 组件启动后，每隔一个固定时间段，触发相同的事件。

17.1.1 Timer 组件有什么用

开发人员可以通过使用 System.Windows.Forms 的 Timer 组件定期地引发指定的某一事件。例如，如果开发者想创建一个过程，使其以特定时间间隔运行直至一个循环完成，或者使其在经过所设置的时间间隔后运行，通过使用 Timer 组件便可使这种过程成为可能。

说明：在 .NET 中，一共提供了 3 种定时器，分别是 Forms.Timer、Timers.Timer 和 Thread.Timer，这 3 种定时器既有相同之处，也有不同之处。在本节中将重点介绍命名空间是 System.Windows.Forms 的 Timer 组件，剩下的两个类将在后面的章节中进行介绍。

Timer 组件在 Visual Studio 工具箱中的图示如图 17.1 所示，将 Timer 组件从工具栏拖曳到 Windows 应用程序的窗体时，相应的 Timer 组件将出现在 Windows 窗体设计器窗口的下方的组件托盘区，如图 17.2 所示。

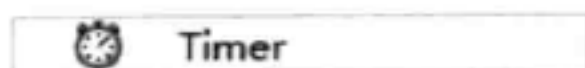


图 17.1 Timer 组件

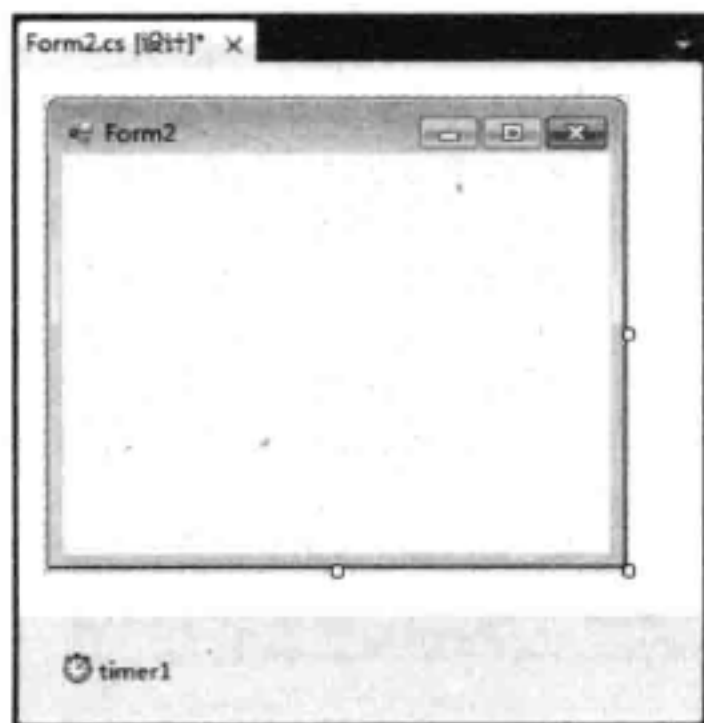


图 17.2 Timer 组件

17.1.2 用 Timer 控制程序定时执行

在窗体中添加了 Timer 组件后,便可以通过对 Timer 组件的相关属性和方法进行设置,从而完成应用程序的功能。在使用 Timer 组件时,通常是用来设置一个时间间隔,每隔一定的时间间隔,则引发事件,执行用户指定的事件处理程序,这个时间间隔便是由 Interval 属性来设置的。Interval 属性是一个 int 类型的属性值,使用此属性值能以毫秒为单位计算时间间隔。可以使用如下代码对 Interval 属性进行设置:

```
private void setTimerInterval(int interval)
{
    //设置计时器的间隔
    this.timer1.Interval = interval;
}
```

而引发的事件则是 Timer 组件的 Tick 事件,开发人员可以在 Tick 事件中添加事件处理程序,则此处理程序在每个时间间隔到来时被运行一次。

在下面的程序中,将通过对 Timer 组件的 Interval 属性和 Tick 事件的设置,使一个标签控件正确地显示当前时间,代码如下:

```
public class Form1 : Form
{
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Timer timer1;
    public Form1()
    {
        InitializeComponent();
        //设置窗体中的控件和组件
        this.timer1 = new System.Windows.Forms.Timer(this.components);
        this.label1 = new System.Windows.Forms.Label();
        //初始化 timer1
        this.timer1.Enabled = true;
        //将时间间隔设置为 1 秒
        this.timer1.Interval = 1000;
        this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
        //初始化 label1
        this.label1.AutoSize = true;
        this.label1.Font = new System.Drawing.Font("SimSun", 17.75F, System.
            Drawing.FontStyle.Regular,
            System.Drawing.GraphicsUnit.Point,
            ((byte) 134));
        this.label1.Location = new System.Drawing.Point(25, 20);
        this.label1.Name = "label1";
        this.label1.Size = new System.Drawing.Size(0, 20);
        this.label1.TabIndex = 0;
        //将 label1 控件添加到窗体上
        this.Controls.Add(this.label1);
        this.Name = "Form1";
        this.Text = "当前时间";
    }
    //当计时器的时间间隔到达时发生
    private void timer1_Tick(object sender, EventArgs e)
    {
        //将 label1 的文本设置为当前时间
        this.label1.Text = DateTime.Now.ToString();
    }
}
```



```

    }
}

```

在上面的程序中，每当程序触发定时器组件的 Tick 事件时，将会引发 Tick 事件的处理函数，此事件的处理函数会将窗体中的标签控件的文本内容设置为系统的当前时间。

注意：程序中使用 DateTime 结构的 Now 属性获得系统的当前时间。DateTime 结构是一个用以表示时间的类型，使用此结构的相关方法和属性，可以对从公元 0001 年 1 月 1 日午夜 12:00:00 到公元 9999 年 12 月 31 日晚上 11:59:59 之间的日期和时间进行获取和操作。具体的使用方法，读者可参见 MSDN 的帮助文档。

对程序进行编译并运行，结果如图 17.3 所示。

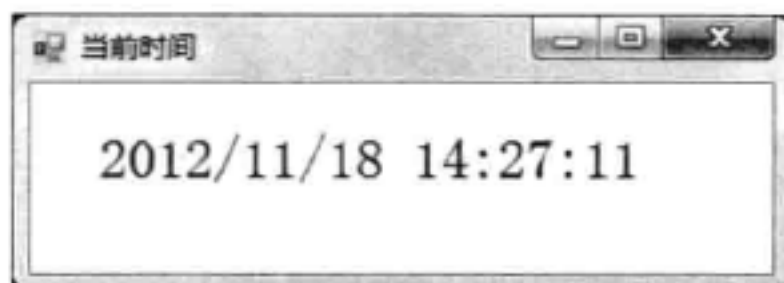


图 17.3 运行结果

在上面的示例代码中，还使用如下语句对 Timer 组件的 Enabled 属性进行设置：

```
this.timer1.Enabled = true;
```

说明：Enabled 属性用来指定 Timer 组件是否可用，其是一个 bool 类型的变量，默认值为 false。可以通过将该属性设置为 true 来启动 Timer 组件，触发 Tick 事件。

17.1.3 用 Timer 制作秒表

下面的代码将设计一个具有秒表功能的窗体。在窗体中，有一个文本标签，此标签用来显示计时开始所运行的秒钟。另外还有两个按钮，第一个按钮用来启动和暂停秒表计时，第二个按钮用来停止并重置秒表。具体的实现代码如下：

```

public class Form1 : Form
{
    //设置窗体中的控件和组件
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Timer timer1;
    //构造函数
    public Form1()
    {
        //设置窗体中的控件和组件
        this.button1 = new System.Windows.Forms.Button();
        this.button2 = new System.Windows.Forms.Button();
        this.label1 = new System.Windows.Forms.Label();
        this.timer1 = new System.Windows.Forms.Timer(this.components);
        //初始化 button1
        this.button1.Location = new System.Drawing.Point(17, 59);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(75, 23);
    }
}

```

```

this.button1.TabIndex = 0;
this.button1.Text = "启动";
this.button1.UseVisualStyleBackColor = true;
this.button1.Click += new System.EventHandler(this.button1_Click);
//初始化 button2
this.button2.Location = new System.Drawing.Point(137, 59);
this.button2.Name = "button2";
this.button2.Size = new System.Drawing.Size(75, 23);
this.button2.TabIndex = 1;
this.button2.Text = "停止";
this.button2.UseVisualStyleBackColor = true;
this.button2.Click += new System.EventHandler(this.button2_Click);
//初始化 label1
this.label1.AutoSize = true;
this.label1.Font = new System.Drawing.Font("SimSun", 21.75F, System.
    Drawing.FontStyle.Regular,
    System.Drawing.GraphicsUnit.Point,
    ((byte) (134)));
this.label1.ForeColor = System.Drawing.Color.Red;
this.label1.Location = new System.Drawing.Point(99, 9);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(28, 29);
this.label1.TabIndex = 2;
this.label1.Text = "0";
初始化 timer1
this.timer1.Interval = 1000;
this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
//将控件添加到窗体中
this.Controls.Add(this.label1);
this.Controls.Add(this.button2);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "秒表";
}
//当计时器的时间间隔到达时发生
private void timer1_Tick(object sender, EventArgs e)
{
    //显示秒表时间
    this.label1.Text = Convert.ToString(Convert.ToInt32(this.label1.
        Text) + 1);
}
//当用户单击 button1 时发生
private void button1_Click(object sender, EventArgs e)
{
    //button1 文本如果为暂停
    if (button1.Text == "暂停")
    {
        //将 button1 文本设为暂停
        button1.Text = "启动";
        //将计时器组件设为不可用
        timer1.Enabled = false;
    }
    //button1 文本如果为启动
    else
    {
        //将 button1 文本设为暂停
        button1.Text = "暂停";
        //启动定时器组件
    }
}


```



```

        timer1.Enabled = true;
    }
}
//当用户单击 button2 时发生
private void button2_Click(object sender, EventArgs e)
{
    button1.Text = "启动";
    timer1.Enabled = false;
    //文本标签中文本归 0
    label1.Text = "0";
}
}

```

 注意：计时器在关闭时重置，并不存在暂停 Timer 组件的方法。


对程序进行编译并运行，结果如图 17.4 所示。



图 17.4 运行结果

17.1.4 启动和重置 Timer

除了对 Timer 组件的 Enable 属性进行设置外，还可以通过使用 Timer 组件的 Start() 和 Stop() 两种方法来打开和关闭计时器。

 注意：使用 Start() 和 Stop() 两种方法与设置 Enable 属性的值所起到的效果相同，都是触发 Timer 组件的 Tick 事件。

在下面的代码中，将通过使用 Start() 和 Stop() 方法来对定时器进行启动和重置，使程序每运行十分钟就会弹出一个消息对话框，询问是否要退出程序，安装更新。

```

public class Class1
{
    static Timer timer1 = new Timer();
    static int counter = 1;
    static bool ifExit = false;
    //当计时器的时间间隔到达时发生
    private static void timer1_Tick(Object myObject, EventArgs myEventArgs)
    {
        //Timer 组件的 Tick 事件被触发后，Timer 组件被重置
        timer1.Stop();
        DialogResult dr = new DialogResult();
        //获取用户响应
        dr = MessageBox.Show("发现可用的更新，是否?",
            "警告次数是：" + alarmCounter, MessageBoxButtons.YesNo);
        if (dr == DialogResult.Yes)
        {
            counter = counter + 1;
        }
    }
}

```

```

        //启动定时器
        timer1.Enabled = true;
    }
    else
    {
        ifExit = true;
    }
}
//程序入口地址
public static void Main()
{
    timer1.Tick += new EventHandler(timer1_Tick);
    //设置时间间隔
    timer1.Interval = 5000;
    //启动定时器
    timer1.Start();
    //若用户选择消息对话框中的“否”按钮，则退出程序
    while(!ifExit)
    {
        Application.Exit();
    }
}
}

```

17.2 图形控件

在.NET 中图形控件主要有 PictureBox 控件和 ImageList 组件两种。本节中，将分别对这两种控件进行详细的介绍。

17.2.1 用 PictureBox 控件显示图片

PictureBox 控件是用于显示图形的控件。在 Windows 应用程序中，有时需要一个显示图形却又不获得焦点的控件，这时，就可以使用 PictureBox 控件。

PictureBox 控件在 Visual Studio 中命名空间为 System.Windows.Forms。它在 Visual Studio 工具箱中的图示如图 17.5 所示，将 PictureBox 控件拖曳到窗体选定后，如图 17.6 所示。下一步，可在 Visual Studio.NET 的属性窗体中对 PictureBox 的属性进行设置，使其达到应用程序的具体需求。

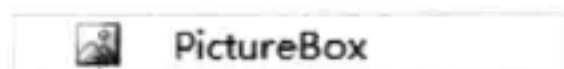


图 17.5 PictureBox 控件



图 17.6 PictureBox 控件

单击 Visual Studio 窗体设计器中的 PictureBox 控件右上方的向右箭头，将打开如图 17.7 所示的 PictureBox 任务对话框，单击任务窗体中的“选择图像”标签项，将弹出如图 17.8

所示的选择资源对话框。在该对话框中可以看出，PictureBox 控件可以从两个方面导入图片，分别是：

- ☐ 本地资源导入。
- ☐ 项目资源导入。

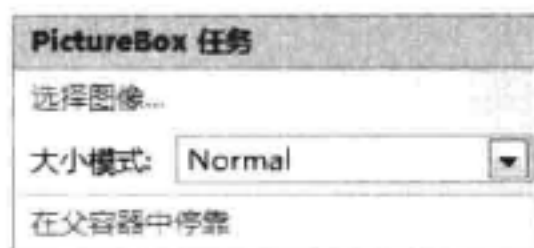


图 17.7 PictureBox 任务窗体



图 17.8 “选择资源”对话框

说明：对于从项目资源中导入的图片，必须先将此图片导入到项目资源中。

图片导入后，开发人员并不能保证所导入的图片大小与 PictureBox 控件的大小完全相同，当 PictureBox 控件无法完整地显示整幅图像时，PictureBox 控件可根据 SizeMode 属性所设置的方式裁剪图像，并在该控件中展示出来。

17.2.2 PictureBox 显示图片的 5 种方式

SizeMode 属性一共提供了 5 种显示图片的方法，在下面将通过具体的例子对这 5 种方法进行说明。在示例程序中，将如图 17.9 所示的图片添加到 Windows 应用程序的 PictureBox 控件中后，由于 SizeMode 的属性值设置的不同，显示的结果也不相同。

说明：当使用 PictureBox 控件时，通常不要求该控件显示出边框，但开发人员同样可以通过设置该控件的 BorderStyle 属性，使该控件使用开发人员所指定的边框并与窗体中的其他控件相分隔。

- ☐ 当 SizeMode 属性值为 PictureBoxSizeMode.Normal 时，添加到 PictureBox 控件中的图像将位于 PictureBox 控件的左上角，当加载的图像比加载它的 PictureBox 控件大时，则该图像将从左上角被剪裁为与 PictureBox 控件大小相等的图片，如图 17.10 所示。
- ☐ 当 SizeMode 属性值为 PictureBoxSizeMode.StretchImage 时，添加到 PictureBox 控件中的图像将自动调整大小，使其适应 PictureBox 控件的大小，将图片完全显示出来，如图 17.11 所示。



图 17.9 示例图片



图 17.10 Normal 方式显示图片



图 17.11 StretchImage 方式显示图片

- ❑ 当 `SizeMode` 属性值为 `PictureBoxSizeMode.AutoSize` 时, `PictureBox` 控件将自动调整大小, 使添加到其中的图片可以在 `PictureBox` 控件中完全显示出来。但是, 当 Windows 应用程序的窗体大小不变时, 所设置的效果与 `SizeMode` 属性值为 `PictureBoxSizeMode.Normal` 时相同, 具体效果如图 17.12 所示。
- ❑ 当 `SizeMode` 属性值为 `PictureBoxSizeMode.CenterImage` 时, 添加到 `PictureBox` 控件中的图像将位于 `PictureBox` 控件的中心, 当加载的图像比加载它的 `PictureBox` 控件大时, 则该图片中心处的图像将在 `PictureBox` 控件中显示出来, 图片的四周边缘将被剪裁, 如图 17.13 所示。
- ❑ 当 `SizeMode` 属性值为 `PictureBoxSizeMode.Zoom` 时, 添加到 `PictureBox` 控件中的图像将按图片原长宽比例进行缩放后, 完全显示在 `PictureBox` 控件中, 如图 17.14 所示。



图 17.12 AutoSize 方式显示图片



图 17.13 CenterImage 方式显示图片



图 17.14 Zoom 方式显示图片

在下面的示例程序中, 将使用代码的方式设置 `PictureBox` 控件中的图片及其显示方式。

```
private void SetPictureBox()
{
    //设置 pictureBox1 的调整大小方式
    pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
    //设置 pictureBox1 的边框类型
    pictureBox1.BorderStyle=BorderStyle.FixedSingle;
    //设置 pictureBox1 的图片
    pictureBox1.Image = Image.FromFile
        (System.Environment.GetFolderPath(System.Environment.
            SpecialFolder.Personal)
            + @"\My Pictures\示例图片\Water lilies.jpg")
}
```

17.2.3 更新 PictureBox 中的图片

如果在程序运行过程中, 希望可以更改 `PictureBox` 控件中所显示的图片, 那么首先要

释放此图片使用的内存，然后清除该控件中的图片，最后向该控件中添加新的要显示的图片。具体代码如下：

```
private void SetPictureBox()
{
    //如果 pictureBox1 的图片不为空
    if (pictureBox1.Image != null)
    {
        //清除 pictureBox1 的图片
        pictureBox1.Image.Dispose();
        pictureBox1.Image = null;
    }
    //设置 pictureBox1 的图片
    pictureBox1.Image = Image.FromFile
        System.Environment.GetFolderPath(System.Environment.
            SpecialFolder.Personal)
        + @"\My Pictures\示例图片\Water lilies.jpg")
}
```

注意：在 Windows 应用程序中，如果有多个 PictureBox 控件加载了同一个图片文件，那么需要将这个图片文件复制与加载它的 PictureBox 控件数目相同的份数，供每个 PictureBox 控件使用，而不能让多个 PictureBox 控件加载同一个文件。

17.2.4 用 ImageList 存储图片

ImageList 组件提供了对 Windows 窗体中所使用到的图像的存储和管理方法，这个图片列表里的图片随后可在 Windows 窗体中的其他控件中引用这些图片。

ImageList 组件在 Visual Studio 中命名空间为 System.Windows.Forms。它在 Visual Studio 工具箱中的图示如图 17.15 所示。ImageList 组件不能添加到窗体中，它将出现在窗体设计器的底部的组件托盘区中，如图 17.16 所示。



图 17.15 ImageList 组件

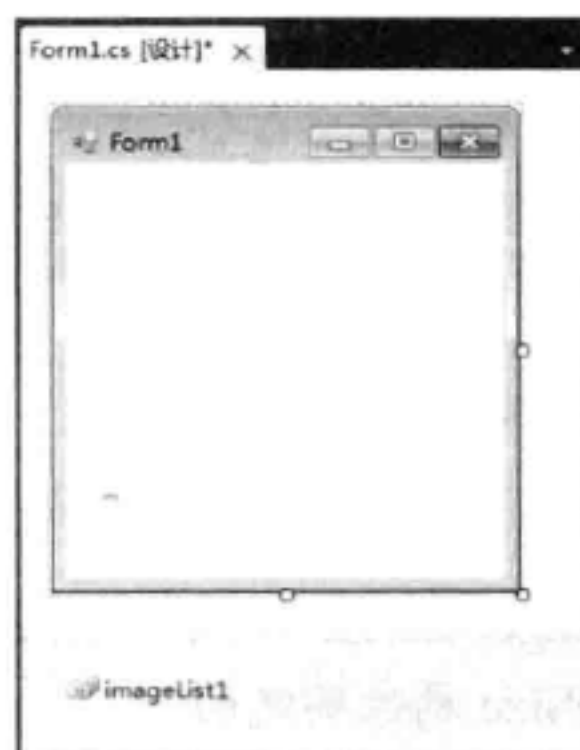


图 17.16 ImageList 组件

对 ImageList 组件的 Images 属性进行编辑，打开相应的“图像集合编辑器”对话框，如图 17.17 所示，在此对话框中添加在应用程序窗体中用到的图片，以方便在以后的编程过程中进行引用。在本例中，将向 ImageList 组件中添加 8 个图像。

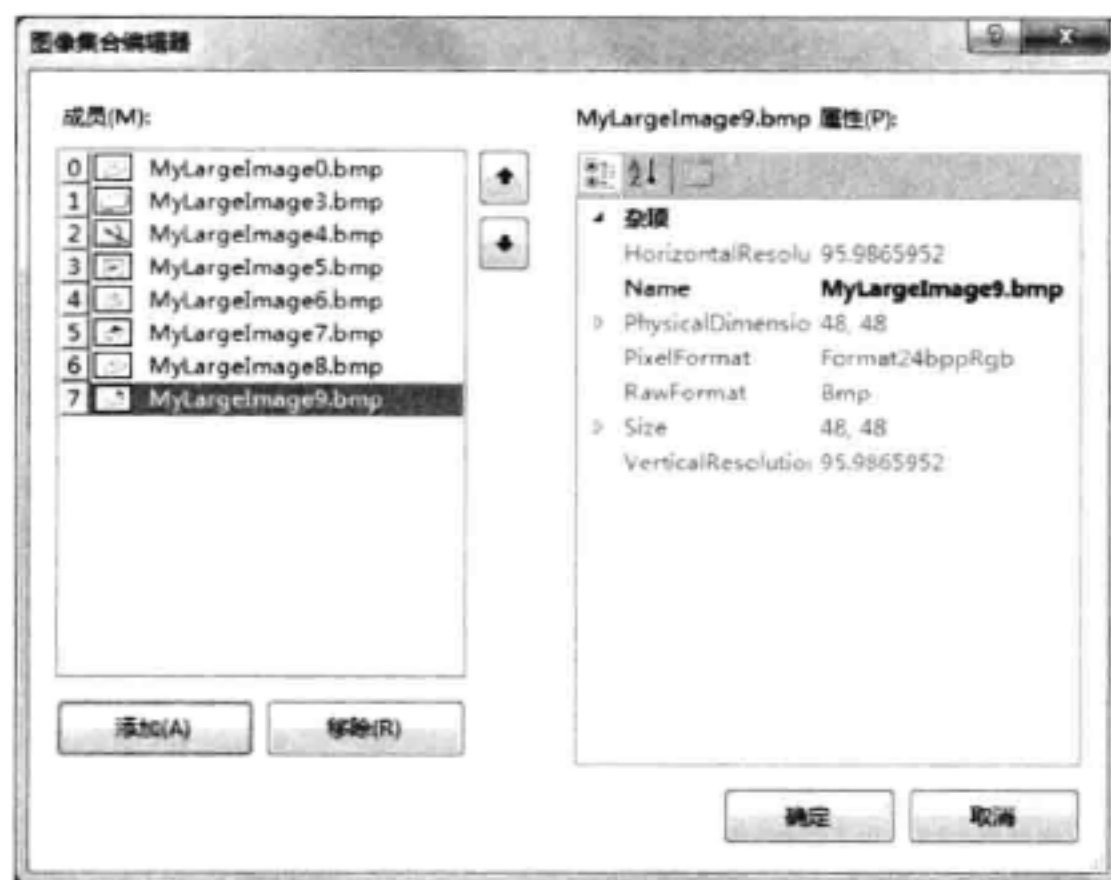


图 17.17 “图像集合编辑器”对话框

也可以使用代码向 ImageList 组件设置图片，具体代码如下：

```
imageList1.ImageStream=(ImageListStreamer)(resources.GetObject("imageLi
st1.ImageStream"));
imageList1.TransparentColor = System.Drawing.Color.Transparent;
imageList1.Images.SetKeyName(0, "MyLargeImage0.bmp");
imageList1.Images.SetKeyName(1, "MyLargeImage3.bmp");
imageList1.Images.SetKeyName(2, "MyLargeImage4.bmp");
imageList1.Images.SetKeyName(3, "MyLargeImage5.bmp");
imageList1.Images.SetKeyName(4, "MyLargeImage6.bmp");
imageList1.Images.SetKeyName(5, "MyLargeImage7.bmp");
imageList1.Images.SetKeyName(6, "MyLargeImage8.bmp");
imageList1.Images.SetKeyName(7, "MyLargeImage9.bmp");
```

而使用代码向 ImageList 组件中添加图片的具体代码如下：

```
public void addImage()
{
    //向 imageList1 添加图片
    System.Drawing.Image myImage = Image.FromFile (@ "C:\ MyLargeImage1.bmp ");
    imageList1.Images.Add(myImage);
}
```

设置好一个 ImageList 组件后，便可以用此组件给使用它的控件的相应属性赋值。对 ImageList 组件中的图像进行访问时，可以通过每个图片的索引值和键值进行访问。

注意：在 ImageList 组件中，所有图片的大小都一致，这个大小是由 ImageList 组件的 ImageSize 属性确定的。

17.2.5 哪些控件可用 ImageList 中的图片

通常来说，ImageList 组件需要与其他控件结合使用，为其他控件提供图片，这些控件主要包括：

- ☐ ListView 控件;
- ☐ TreeView 控件;
- ☐ ToolBar 控件;
- ☐ TabControlButton 控件;
- ☐ CheckBox 控件;
- ☐ RadioButton 控件。

17.3 菜单控件

菜单是通过存放按照一般主题分组的命令将功能公开给用户的控件。.NET 中主要提供 3 种常用的菜单型控件, 分别是 MenuStrip 控件、ToolStrip 控件和 ContextMenuStrip 控件。本节中, 将对这 3 种控件分别进行详细的介绍。

17.3.1 用 MenuStrip 创建菜单

MenuStrip 控件主要用于在 Windows 应用程序中向用户提供常用菜单, 例如 Visual Studio 界面上端的菜单, 如图 17.18 所示。单击这个菜单中的每一项, 都可以执行此项相对应的应用程序操作。

文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) 团队(M) 数据(A) 格式(O) 工具(T) 体系结构(C) 测试(S) 分析(N) 窗口(W) 帮助(H)

图 17.18 Visual Studio 界面菜单

使用 MenuStrip 控件生成的菜单, 可以使开发人员自由地定义各种菜单项及菜单项的子项, 在 .NET 中提供了多种类型的菜单项子项来增加菜单的开发建议性和使用方便性, 这些子项都可以将 MenuStrip 控件作为容器将各菜单子项添加到其中。

MenuStrip 控件的命名空间是 System.Windows.Forms, 它在 Visual Studio 工具箱中的图示如图 17.19 所示。将 MenuStrip 控件拖曳到窗体选定后, 窗体的顶部将出现浅灰色的菜单项, 并在第一项的位置处有“请在此处输入”字样的提示信息出现, 同时在 Visual Studio 窗体设计器底部的组件托盘区将出现一个 MenuStrip 控件的实例, 如图 17.20 所示。下一步, 可在 Visual Studio 属性窗体中对 MenuStrip 控件的属性进行设置, 使其达到应用程序的具体需求。

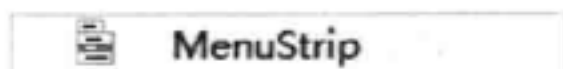


图 17.19 MenuStrip 控件



图 17.20 MenuStrip 控件

17.3.2 有哪些方法添加菜单项

Items 属性是 MenuStrip 控件中最重要的一個属性，用来表示菜单中所有项的集合。对菜单中添加子项可以通过 3 种方法实现，下面将分别针对这 3 种方法进行说明。

1. 使用项集合器进行添加

单击窗体中 MenuStrip 控件右上角的向右箭头，将打开如图 17.21 所示的“MenuStrip 任务”窗体，单击该窗体下方的“编辑项...”文本标签，将打开如图 17.22 所示的“项集合编辑器”对话框。



图 17.21 “MenuStrip 任务”窗体

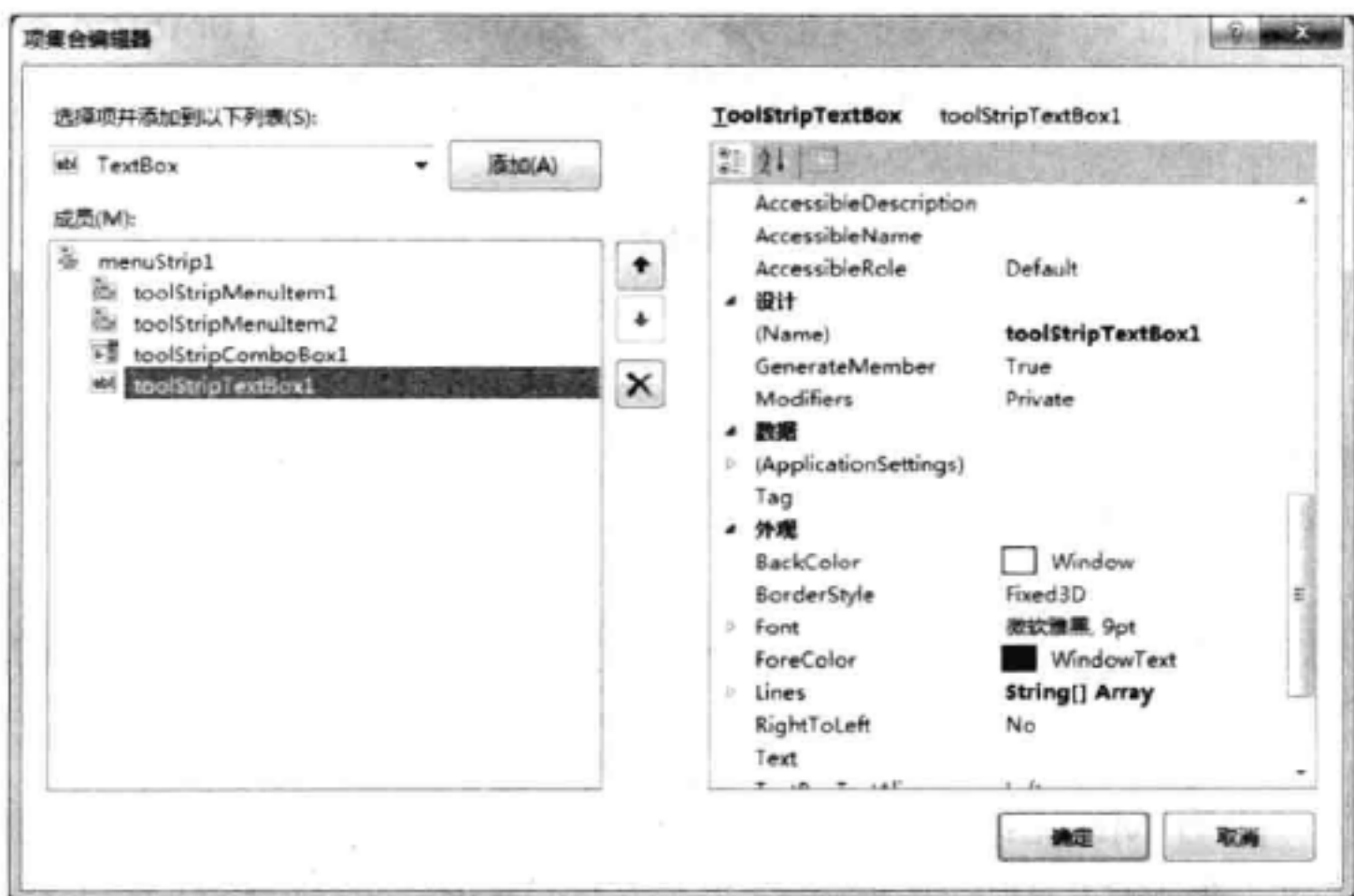


图 17.22 “项集合编辑器”对话框（1）

在此窗体中，可以向 MenuStrip 控件中添加 ToolStripMenuItem、ToolStripComboBox 和 ToolStripTextBox 3 种类型的菜单项，其外观如图 17.23 所示。

当菜单项为 ToolStripComboBox 类型时，开发人员可以通过编辑 ToolStripComboBox 类的 Items 属性向此菜单项的下拉列表中添加候选项。此菜单项的属性设置和相关事件与 ComboBox 控件的属性和事件相类似。

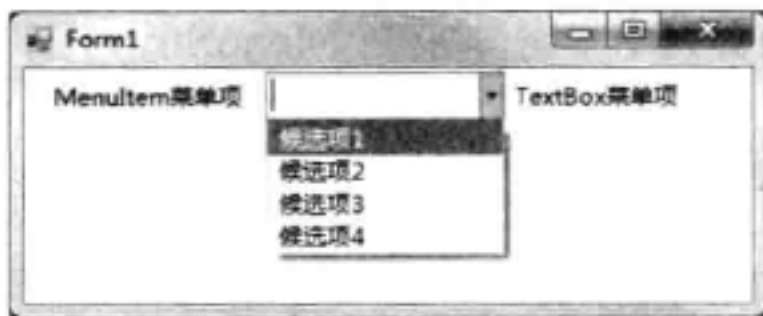


图 17.23 菜单项编辑器

当菜单项为 ToolStripTextBox 类型时，开发人员可以通过编辑 ToolStripComboBox 类的 Text 属性，从而编辑此菜单文本框的文本内容。此菜单项的属性设置和相关事件与 TextBox 控件的属性和事件类似。

但是，在 MenuStrip 控件中，最常用的菜单项还是 ToolStripMenuItem 项，通过编辑 ToolStripMenuItem 项的 DropDownItems 属性，可以向菜单项添加相对应的子项。单击属性窗体中 DropDownItems 属性旁的...，可以打开如图 17.24 所示的菜单子项编辑器。

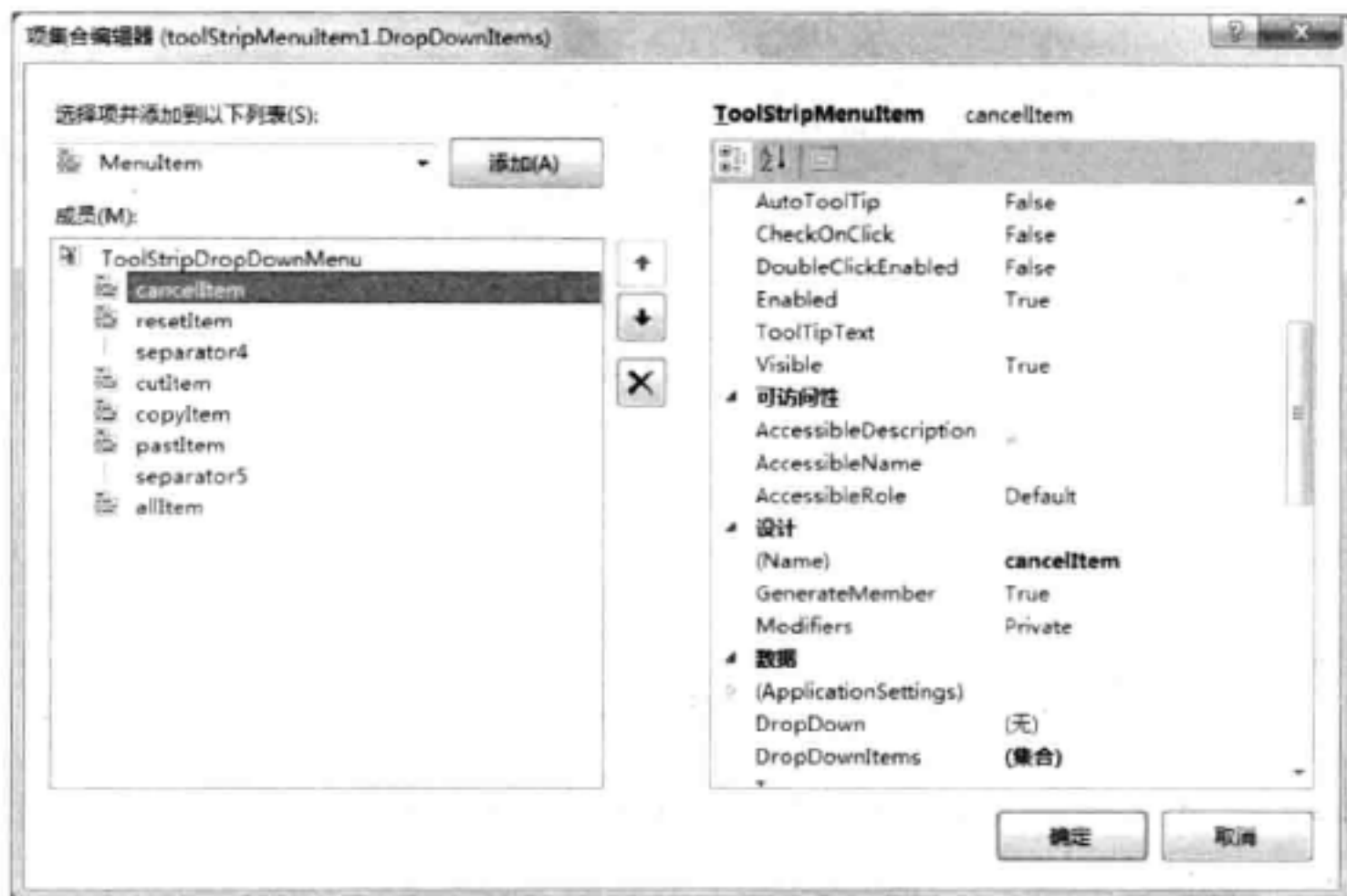


图 17.24 “项集合编辑器”对话框 (2)

说明：ToolStripMenuItem 类还有一个重要的属性就是 Overflow 属性，此属性指出当菜单项有溢出现象时该如何显示。

2. 在窗体中直接添加

除了使用菜单编辑器和子菜单编辑器对菜单进行可视化的设计外，开发人员也可以在 Visual Studio 窗体设计器中对菜单项进行设计。在工具栏中将 MenuItem 控件拖曳到窗体的指定位置，单击相应的 ToolStripMenuItem 项的向下箭头，可以设置 MenuItem 的类型，如图 17.25 所示。

说明：开发者可以通过双击相应的 ToolStripMenuItem 项来添加菜单项并对菜单项文本进行编辑，如图 17.26 所示。

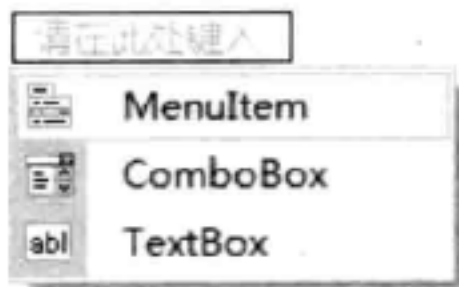


图 17.25 MenuItem 类型设置



图 17.26 MenuItem 项的编辑

3. 使用代码进行添加

有的时候，程序需要根据程序的进行过程动态地生成窗体的菜单。此时，可使用以下程序向 Windows 应用程序的窗体动态地添加菜单及相应的菜单项。

```
//设置窗体里的控件
private System.Windows.Forms.MenuStrip menuStrip1;
private System.Windows.Forms.ToolStripMenuItem fileItem;
```

```

private System.Windows.Forms.ToolStripItem newItem;
private System.Windows.Forms.ToolStripItem openItem;
private System.Windows.Forms.ToolStripItem saveItem;
private System.Windows.Forms.ToolStripItem editItem;
private System.Windows.Forms.ToolStripItem cutItem;
private System.Windows.Forms.ToolStripItem copyItem;
private System.Windows.Forms.ToolStripItem pasteItem;
//初始化控件
public Form1()
{
    InitializeComponent();

    //菜单项
    menuStrip1 = new System.Windows.Forms.MenuStrip();
    //菜单中的“文件”子项
    fileItem = new System.Windows.Forms.ToolStripItem();
    //菜单中的“编辑”子项
    editItem = new System.Windows.Forms.ToolStripItem();
    //菜单中的“新建”子项
    newItem = new System.Windows.Forms.ToolStripItem();
    //菜单中的“打开”子项
    openItem = new System.Windows.Forms.ToolStripItem();
    //菜单中的“保存”子项
    saveItem = new System.Windows.Forms.ToolStripItem();
    //菜单中的“剪切”子项
    cutItem = new System.Windows.Forms.ToolStripItem();
    //菜单中的“复制”子项
    copyItem = new System.Windows.Forms.ToolStripItem();
    //菜单中的“粘贴”子项
    pasteItem = new System.Windows.Forms.ToolStripItem();
    //menuStrip1
    menuStrip1.Items.AddRange(new ToolStripItem[] {fileItem,editItem});
    menuStrip1.Text = "menuStrip1";
    //fileItem
    fileItem.DropDownItems.AddRange(new ToolStripItem[] {newItem,openItem,
    saveItem});
    fileItem.Text = "文件";
    //editItem
    editItem.DropDownItems.AddRange(new ToolStripItem[] {cutItem,copyItem,
    pasteItem});
    editItem.Text = "编辑";
    //fileItem的子项
    newItem.Text = "新建";
    openItem.Text = "打开";
    saveItem.Text = "保存";
    //editItem的子项
    cutItem.Text = "剪切";
    copyItem.Text = "复制";
    pasteItem.Text = "粘贴";
    //Form1
    this.Controls.Add(this.menuStrip1);
    this.MainMenuStrip = this.menuStrip1;
}

```

编译并运行程序，生成的菜单外观如图 17.27 所示。



图 17.27 程序运行结果

如果开发者希望向窗体添加的是一个标准菜单项，可以单击如图 17.21 所示的 MenuStrip 任务窗体中的“插入标准项”标签项。插入的标准菜单项及其各项的子项列表如图 17.28 所示。

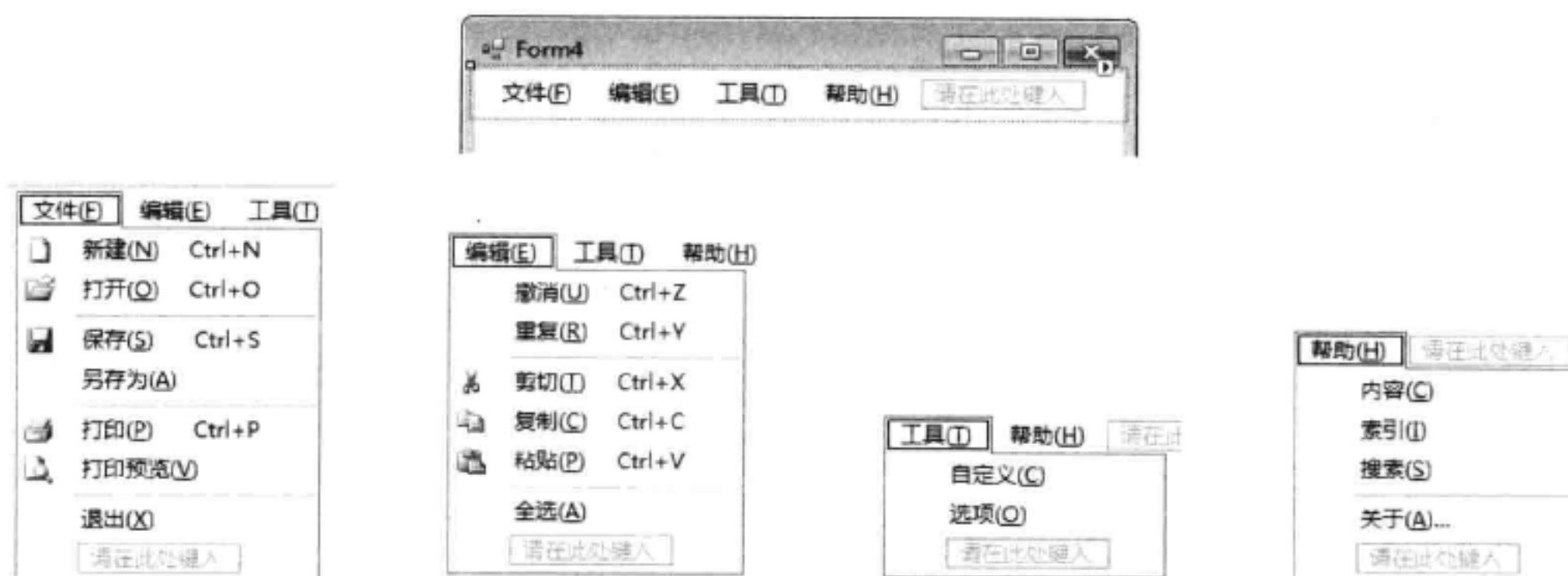



图 17.28 标准菜单项子项列表

 **技巧：**开发人员可以通过对标准菜单项进行编辑来实现应用程序的菜单。

17.3.3 为菜单项编写代码完成相应功能

在对 Windows 窗体中添加了指定的菜单项后，可以通过指定菜单的属性和行为增加菜单的功能。

在下面的应用程序示例中，将通过单击在菜单上的“窗体|新建”项来打开新的应用文件子窗体，具体代码如下所示。

```
public class Form1 : Form
{
    private System.Windows.Forms.MenuStrip menuStrip1;
    private System.Windows.Forms.ToolStripMenuItem WindowMenuItem;
    private System.Windows.Forms.ToolStripMenuItem NewMenuItem;
    public Form1()
    {
        InitializeComponent();
        menuStrip1 = new System.Windows.Forms.MenuStrip();
        WindowMenuItem = new System.Windows.Forms.ToolStripMenuItem();
        NewMenuItem = new System.Windows.Forms.ToolStripMenuItem();
        //初始化菜单项
        menuStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[]
        {WindowMenuItem});
        menuStrip1.Location = new System.Drawing.Point(0, 0);
    }
}
```

```

menuStrip1.MdiWindowListItem = this.WindowMenuItem;
menuStrip1.Name = "menuStrip1";
menuStrip1.Size = new System.Drawing.Size(292, 24);
menuStrip1.TabIndex = 0;
menuStrip1.Text = "menuStrip1";
//初始化 WindowMenuItem
WindowMenuItem.DropDownItems.AddRange(new System.Windows.Forms.
Tool StripItem[] {
NewMenuItem});
WindowMenuItem.Name = "WindowMenuItem";
WindowMenuItem.Size = new System.Drawing.Size(41, 20);
WindowMenuItem.Text = "窗体";
//初始化 NewMenuItem
NewMenuItem.Name = "NewMenuItem";
NewMenuItem.Size = new System.Drawing.Size(172, 22);
NewMenuItem.Text = "新建";
NewMenuItem.Click += new System.EventHandler(this.NewMenuItem_Click);
//将菜单添加到窗体中
this.IsMdiContainer = true;
this.Controls.Add(this.menuStrip1);
}
//单击 NewMenuItem 时引发
private void NewMenuItem_Click(object sender, EventArgs e)
{
    Form newForm = new Form();
    //为子窗体设置父窗体
    newForm.MdiParent = this;
    //显示新窗体
    newForm.Show();
}
}

```

在上面的代码中, 通过将主窗体的 `IsMdiContainer` 属性设置为 `true`, 指示此窗体是一个多文档窗体的容器, 并将一个 `MenuStrip` 控件添加到窗体中, 向菜单控件中添加一个文本内容为“窗体”的子菜单, 并向此子菜单中添加一个名为“新建”的子菜单项。使用以下代码将新增菜单项的 `MdiWindowListItem` 属性设置为向此菜单添加的“窗体”子菜单。那么当作为多文档窗体的容器的窗体产生一个新的子窗体时, 这个新产生的子窗体名称将被自动添加到此子菜单项下方。

```
this.menuStrip1.MdiWindowListItem = this.WindowMenuItem;
```

最后, 添加 `NewMenuItem` 的 `Click` 事件处理程序。在该事件处理程序内插入下面的代码, 那么在单击“新建”子项时, 将创建并显示一个 `Form` 类的新实例。

```

private void NewMenuItem_Click(object sender, EventArgs e)
{
    Form newForm = new Form();
    //为子窗体设置父窗体
    newForm.MdiParent = this;
    //显示新窗体
    newForm.Show();
}

```

对上述程序进行编译并运行, 运行结果如图 17.29 所示。单击菜单中的“窗体”|“新建”命令将会在多文档窗体容器中打开新的窗体, 并将此窗体名称添加到“窗体”菜单项

中。用户可以通过在菜单中选择不同的项来激活不同的窗体。

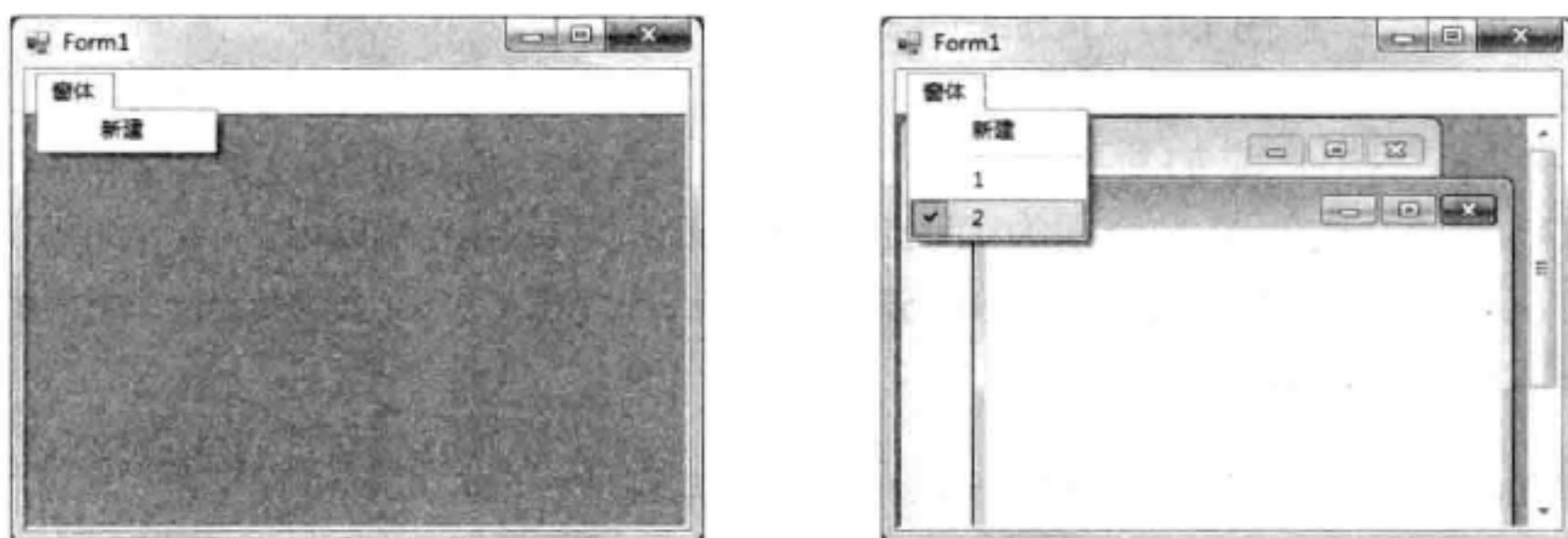



图 17.29 运行界面

 说明：当应用程序是 MDI 应用程序时，通常还需要添加合并菜单项的功能。

17.3.4 用 ToolStrip 创建工具栏

ToolStrip 控件是用于向 .NET 开发的 Windows 应用程序添加工具栏的控件。工具栏控件由一组图形按钮组成，例如 Visual Studio 界面上端的工具栏，如图 17.30 所示。一般来说，工具栏中的项都是窗体菜单中的各常用项，如新建、打开和复制等。

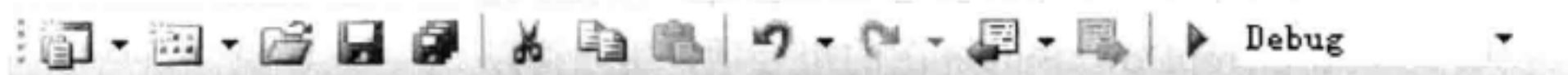


图 17.30 Visual Studio 工具栏

ToolStrip 控件的命名空间是 System.Windows.Forms，它在 Visual Studio 工具箱中的图示如图 17.31 所示，将 ToolStrip 控件拖曳到窗体选定后，窗体的顶部将出现浅灰色的工具栏项，并在第一项的位置处出现一个用于提示用户添加新项的图标。同时在 Visual Studio 窗体设计器的底部将出现一个 ToolStrip 控件的实例，如图 17.32 所示。下一步，可在 Visual Studio 属性窗体中对 ToolStrip 控件的属性进行设置，使其达到应用程序的具体需求。



图 17.31 ToolStrip 控件

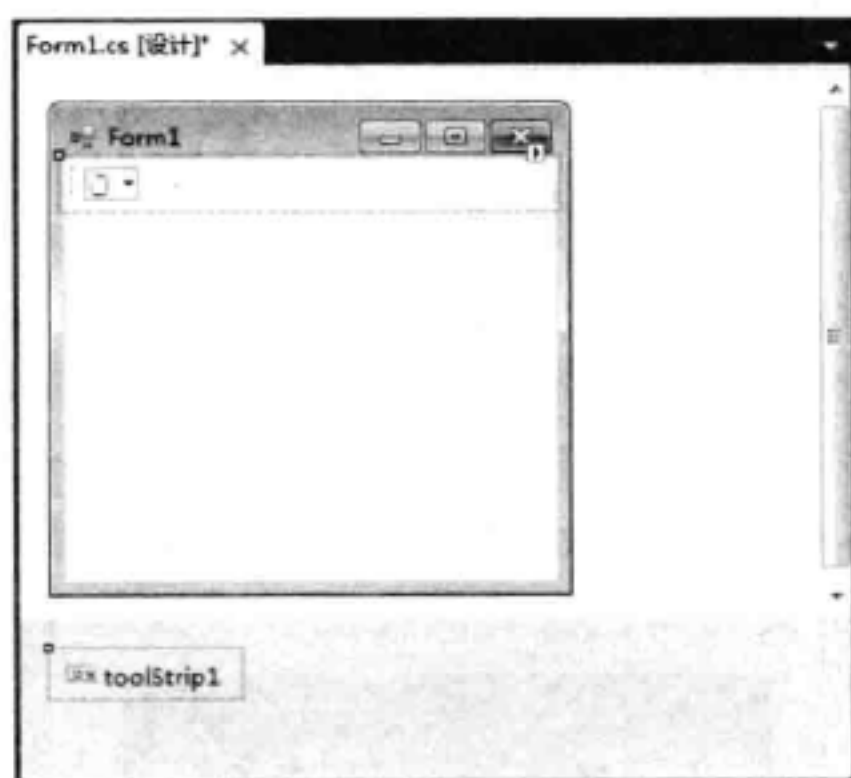



图 17.32 ToolStrip 控件

17.3.5 工具栏中可使用的按钮类型

单击用于提示用户添加新项的图标右侧的向下箭头，可以设置 `ToolStripItem` 的类型，如图 17.33 所示。这些类型分别为 `ToolStripButton`、`ToolStripLabel`、`ToolStripDropDownButton`、`ToolStripSplitButton`、`ToolStripSeparator`、`ToolStripComboBox`、`ToolStripTextBox` 和 `ToolStripProgressBar`。

 说明：`ToolStrip` 控件是各 `ToolStripItem` 项的容器。

`ToolStrip` 控件中可添加的 `ToolStripItem` 类型的具体作用如下所述。

- ☐ `ToolStripLabel` 项可以在工具栏中提供类似于标签控件的功能，用于向用户显示提示性信息。
- ☐ `ToolStripTextBox` 项可以在工具栏中提供类似于文本框控件的功能，用于获取用户输入的信息。
- ☐ `ToolStripButton` 项可以在工具栏中提供类似于按钮控件的功能，允许用户通过单击来执行指定的操作。
- ☐ `ToolStripDropDownButton` 项是一个下拉控件项，类似于菜单中的一个子项，用户可以通过单击该菜单旁的下拉按钮，在下拉菜单中选择一项。
- ☐ `ToolStripSplitButton` 项可以看成是 `ToolStripButton` 项和 `ToolStripDropDownButton` 项的组合项，用户可以直接通过单击来执行操作，也可以单击项旁的下拉按钮来选择一项。
- ☐ `ToolStripProgressBar` 项可以在工具栏中提供类似于进度条的功能，用于向用户显示指定操作的进度。
- ☐ `ToolStripComboBox` 项可以在工具栏中提供类似于下拉列表框的功能，用于使用户在下拉列表框中选择希望的数据。
- ☐ `ToolStripSeparator` 项可以在工具栏中提供项的分组，以及项之间的区分。

单击 `ToolStrip` 控件右上角的向右箭头，将打开如图 17.34 所示的“`ToolStrip` 任务”对话框。开发人员可以在此窗体中对 `ToolStrip` 控件的外观和各个子项进行相应的设置。

在“`ToolStrip` 任务”对话框中，开发人员可以通过对 `RenderMode` 属性的设置来设置 `ToolStrip` 控件的绘制样式。当 `RenderMode` 的属性值被设置为 `ToolStripRenderMode.System` 时，使用系统颜色和平面视觉样式处理 `ToolStrip` 对象的绘制功能，如图 17.35 所示；当 `RenderMode` 的属性值被设置为 `ToolStripRenderMode.Professional` 时，窗体中工具栏通过应用自定义的调色板和简化的样式处理 `ToolStrip` 对象的绘制功能，如图 17.36 所示。



图 17.33 `ToolStrip` 控件



图 17.34 `ToolStrip` 任务窗体



图 17.35 `ToolStripRenderMode.System`



图 17.36 `RenderMode` 属性设置

另外,开发人员还可以通过在“ToolStrip 任务窗体”对话框中设置 GripStyle 属性,来设置此工具栏中的移动手柄是否可见。当 GripStyle 属性值为 Hidden 时,程序外观如图 17.37 所示;当 GripStyle 属性值为 Visible 时,程序外观如图 17.38 所示。

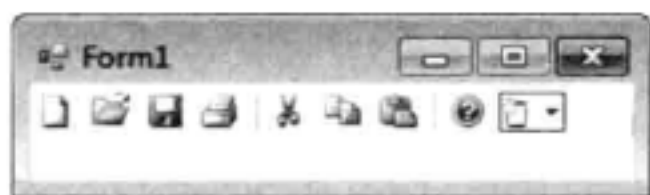


图 17.37 GripStyle 属性设置

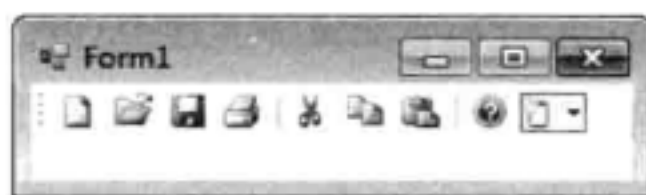


图 17.38 GripStyle 属性设置

说明: 当移动手柄可见时,开发人员可以通过设置相应的代码使用户可以拖动移动手柄来移动工具栏的水平显示位置。

单击任务窗体下方的“编辑项...”文本标签,将打开如图 17.39 所示的“项集合编辑器”对话框。开发人员可以通过此窗体对工具栏中的项进行添加修改和编辑。

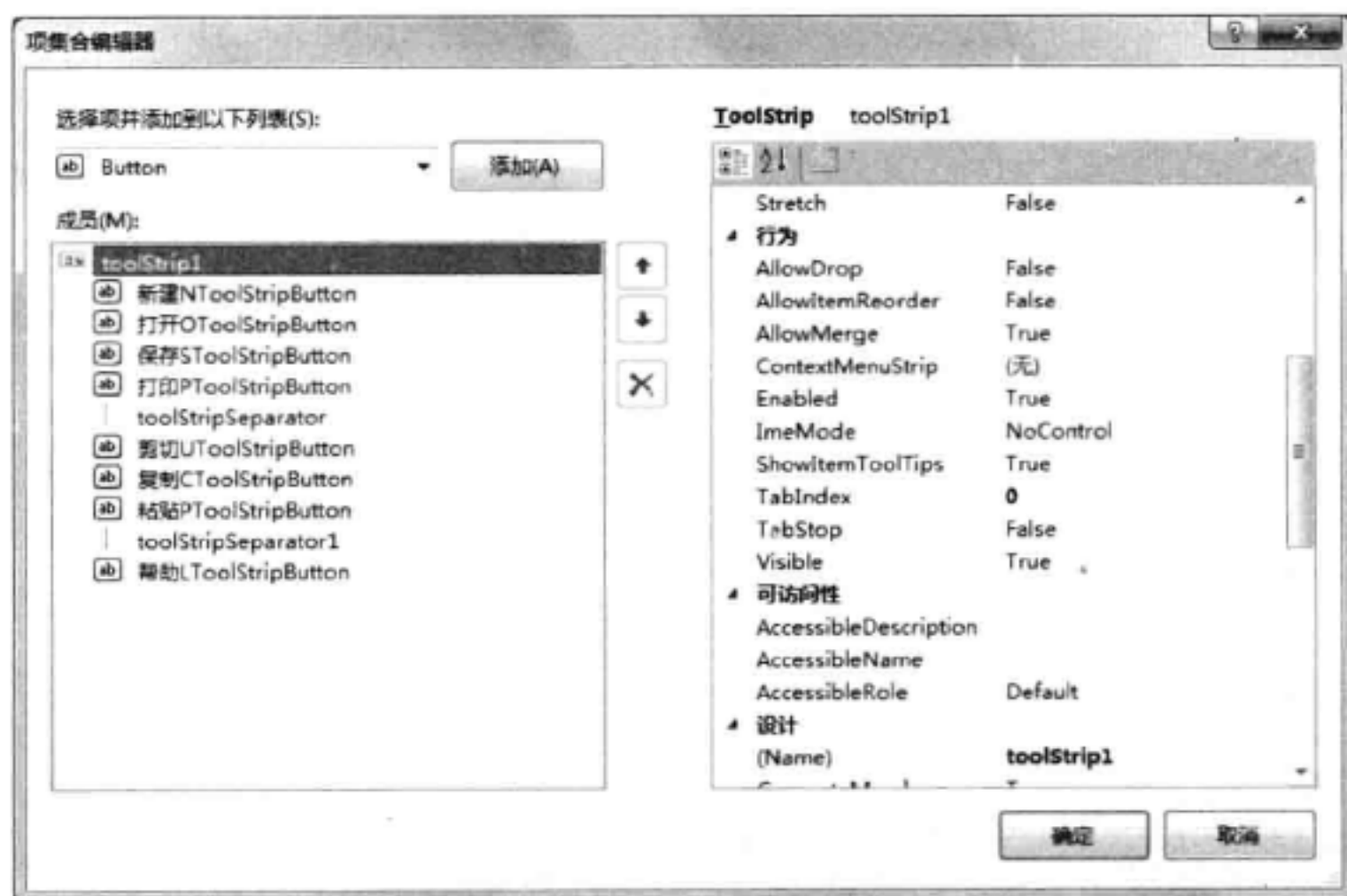


图 17.39 “项集合编辑器”对话框

17.3.6 设置工具栏的外观和行为


在对 Windows 应用程序的工具栏控件添加了子项后,可以通过设置 ToolStrip 控件的相关属性来设置工具栏的外观和行为。下面将通过具体的程序来讲解 ToolStrip 控件的相关属性、方法和事件。

当用户希望可以对 Windows 应用程序窗体的工具栏中的各项进行重新排序时,可以通过将 ToolStrip 控件的 AllowItemReorder 属性设置为 true 来实现。此时,用户可以通过按 Alt 键和鼠标左键来拖动工具栏中各项的相对位置,代码如下:

```
private void setAllowItemReorder()
{
    toolStrip1.AllowItemReorder = true;
}
```

在工具菜单中,当工具菜单的总长度小于所有工具菜单中各项长度的和时,在工具菜

单的最右侧可以出现一个下拉箭头，用户单击此下拉箭头，将打开一个下拉菜单，在里面显示了所有不能在工具栏菜单中显示的项。

 **说明：**用户可以像操作工具栏中的子项一样操作下拉菜单中的项。

ToolStrip 类的 CanOverflow 属性指定了在工具栏中的子项长度和大于工具栏长度时，工具栏是否显示一个下拉按键。当 CanOverflow 属性为 true 时，程序将自动实例化一个 ToolStripOverflowButton 类，并将它加载到工具栏中。如果 CanOverflow 属性值为 false，那么当工具栏中的子项长度和大于工具栏长度时，工具栏中无法显示出来的项就不会再发挥作用，将无法响应用户的操作。

当工具栏溢出发生时，还需要通过指定工具栏中各子项的 Overflow 属性，指出当溢出发生时，各子项的行为。Overflow 属性的属性值是一个 ToolStripItemOverflow 类型的枚举类型值，一共有 3 个枚举值，分别是 Always、Never 和 AsNeeded。

- ❑ 当工具栏中菜单项的 Overflow 属性值为 ToolStripItemOverflow.Always 时，无论工具栏溢出是否发生，此项都只出现在工具栏的溢出菜单中。
- ❑ 当工具栏中菜单项的 Overflow 属性值为 ToolStripItemOverflow.Never 时，无论工具栏溢出是否发生，此项都只出现在工具栏中。
- ❑ 当工具栏中菜单项的 Overflow 属性值为 ToolStripItemOverflow.AsNeeded 时，此项会根据是否出现溢出来决定发生的位置。

在下面的示例程序中，将指定一个工具栏控件的 CanOverflow 属性和每一个子项的 Overflow 属性，使工具栏菜单发生溢出时，工具栏中的溢出项可以添加到工具栏的下拉菜单中。

```
private void setToolStripOverflow()
{
    //设置在工具栏中，允许发生溢出现象
    toolStrip1.CanOverflow = true;
    //对各个子项的溢出行为进行设置
    newToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
    openToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
    saveToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
    printToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
    cutToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
    copyToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
    pasteToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
    helpToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.AsNeeded;
}
```

对上述程序进行编译运行后，得到的结果如图 17.40 所示，改变窗体的大小，则工具栏将出现溢出现象。此时，工具栏中无法正常显示的项将出现在工具栏的下拉菜单中，如图 17.41 所示。



图 17.40 运行结果

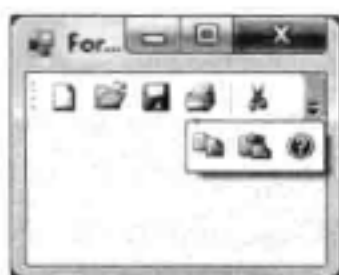


图 17.41 运行结果

当将 `helpToolStripItem` 项的 `Overflow` 属性使用如下语句设置为 `ToolStripItemOverflow.Never` 时,那么在溢出发生时,此工具栏子项将不会自动添加到下拉菜单中,如图 17.42 所示。

```
this.helpToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.Never;
```

当将 `helpToolStripItem` 项的 `Overflow` 属性使用如下语句设置为 `ToolStripItemOverflow.Always` 时,那么无论溢出是否发生,此工具栏子项都只会出现在下拉菜单中,如图 17.43 所示。

```
this.helpToolStripItem.Overflow = System.Windows.Forms.ToolStripItemOverflow.Always;
```

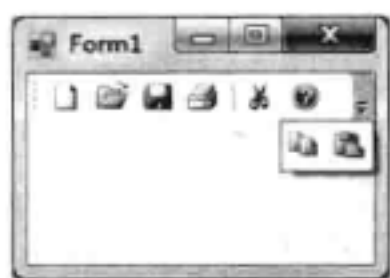


图 17.42 运行结果

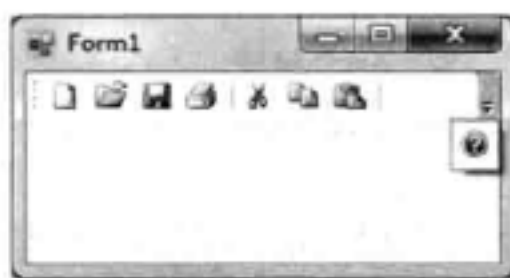



图 17.43 运行结果

17.3.7 设置 ToolStripComboBox 的自动完成功能

在下面的程序中,将向窗体中添加一个 `ToolStrip` 控件,并在此工具栏控件中添加了一个 `ToolStripLabel` 控件和一个 `ToolStripComboBox` 控件,并通过对 `ToolStripComboBox` 控件的 `AutoCompleteMode` 属性和 `AutoCompleteCustomSource` 属性进行设置来启动该控件的自动完成功能。

 说明: 此自动完成功能处理 `ToolStripComboBox` 控件的方式与 `ComboBox` 控件相同。如果用户输入一个与列表中某项的第一个字符匹配的字符,该项将会立即显示。

```
private System.Windows.Forms.ToolStrip toolStrip1;
private System.Windows.Forms.ToolStripLabel toolStripLabel1;
private System.Windows.Forms.ToolStripComboBox toolStripComboBox1
private void setAutoCompleteMode()
{
    toolStrip1 = new System.Windows.Forms.ToolStrip();
    toolStripLabel1 = new System.Windows.Forms.ToolStripLabel();
    toolStripComboBox1 = new System.Windows.Forms.ToolStripComboBox();
    toolStrip1.Items.AddRange(new ToolStripItem[] { toolStripLabel1, toolStripComboBox1 });
    toolStripLabel1.Text = "可选项";
    toolStripLabel1.Overflow = ToolStripItemOverflow.Never;
    toolStripComboBox1.Overflow = ToolStripItemOverflow.Never;
```

```

toolStripComboBox1.AutoCompleteCustomSource.AddRange(new string[] {"第
一项","第二项","第三项"});
toolStripComboBox1.Items.AddRange(new object[] {"第一项","第二项","第三项"});
toolStripComboBox1.AutoCompleteMode = AutoCompleteMode.Append;
}

```

在工具栏菜单的各子项控件中,如果将 `AutoSize` 属性设置为 `true`,那么此子项控件可以为了显示其中的完整内容而自动调节大小。如果将 `AutoSize` 属性设置为 `false`,开发人员可以通过设置子项的 `Margin` 属性来设置此项和相邻项的间距。

17.3.8 设置 ToolStrip 的外观

在下面的代码示例中,设置了 `ToolStrip` 控件中的各 `ToolStripItem` 控件间的相对间距,并指定了各 `ToolStripItem` 项相对于 `ToolStrip` 控件的布局。


```

private void setAutoCompleteMode()
{
    toolStripButton1.AutoSize = false;
    //按左、上、右和下的顺序指定与相邻项的间距
    toolStripButton1.Margin = new System.Windows.Forms.Padding(2, 2, 2, 2);
    //将 toolStripButton1 的 Alignment 属性设置为 Right,则此项与 ToolStrip 的右侧
    对齐
    toolStripButton1.Alignment = System.Windows.Forms.ToolStripItemAlignment.
    Right;
}

```

17.3.9 用 ContextMenuStrip 创建快捷菜单

`ContextMenuStrip` 控件主要用于在 Windows 应用程序中,向用户提供快捷菜单,例如 Visual Studio 窗体设计器中的快捷菜单,如图 17.44 所示。使用快捷菜单可以快速地给出指针所在位置应用程序所对应的快捷操作选项。

 **说明:** 在 .NET 中, `ContextMenuStrip` 控件可以与很多控件相关联使用。创建了 `ContextMenuStrip` 类的实例后,可以将指定控件的 `ContextMenuStrip` 属性设置为此 `ContextMenuStrip` 实例,那么在应用程序运行时,用户在指定控件中单击右键时,将弹出此开发人员定义的快捷菜单。

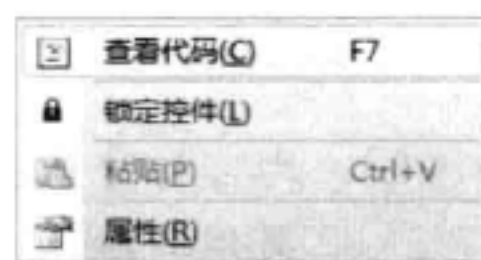


图 17.44 Visual Studio 界面快捷菜单

`ContextMenuStrip` 控件的命名空间是 `System.Windows.Forms`,它在 Visual Studio 工具箱中的图示如图 17.45 所示,将 `ContextMenuStrip` 控件拖曳到窗体选定后,窗体的顶部将出现浅灰色的快捷项,并在第一项的位置处有“请在此处输入”字样的提示信息出现。同时在 Visual Studio 窗体设计器的底部的组件托盘区将出现一个 `ContextMenuStrip` 控件的实例,如图 17.46 所示。下一步,可在 Visual Studio 属性窗体中对 `ContextMenuStrip` 控件的属性进行设置,使其达到应用程序的具体需求。



图 17.45 ContextMenuStrip 控件



图 17.46 ContextMenuStrip 控件

对 Windows 应用程序的快捷菜单中的项进行设置，可以通过如图 17.47 所示的项集合编辑器来完成，具体方法与 MenuStrip 控件中的添加子菜单项的方法相同，在此处不再赘述。

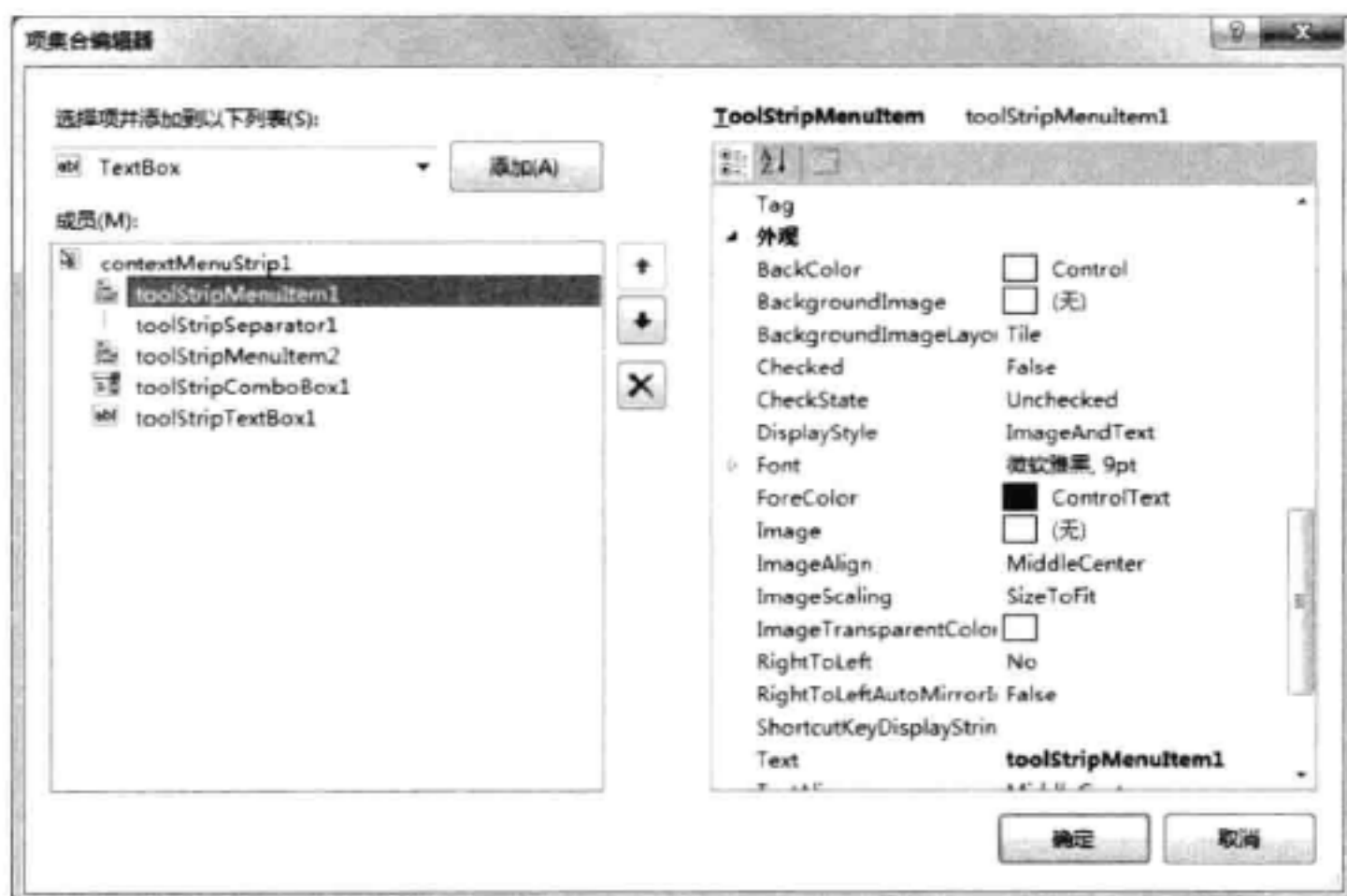


图 17.47 ContextMenuStrip 控件


17.4 本章总结

在本章中，就前面几章没有讲解到的却在 Windows 应用程序编程中广泛应用的控件进行了较为详细的说明，主要分为计时器控件、图形控件和菜单控件。经过本章及前面几章的学习，相信读者已经可以独立地开发一些简单的 Windows 应用程序了。

17.5 实战练习

1. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，向窗体中添加一个 Timer

控件，通过该控件每 10 秒弹出一个问候语。要求在程序中设置多条问候语，每次随机显示一条。

提示：可用 String 数组保存多条问候语，然后使用 Random 随机选出其中的一条。

2. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在窗体中添加一个 ImageList 控件，向该控件中添加多张图片，然后再添加一个 PictureBox 控件和一个 Button 控件，当单击按钮时，在 PictureBox 控件中循环显示 ImageList 控件中的图片。

3. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在 Form1 中创建一个菜单栏，模拟 Windows 中的画图程序的菜单。

4. 接第 3 题，在 Form1 中添加一个工具条，在工具条中添加与菜单中对应的命令按钮。

第 18 章 Windows 应用程序的部署

Windows 应用程序的部署就是把 Windows 应用程序安装到目标系统上的过程。在 .NET 中，主要提供两种 Windows 应用程序部署方案：ClickOnce 部署方案和 Windows Installer 部署方案。在本章中，将就这两种方案进行细致的比较和说明。

18.1 你了解这两种部署方案吗

Visual Studio 为部署基于 Windows 的应用程序提供两种不同的策略：

- ☐ 使用 ClickOnce 技术发布应用程序；
- ☐ 使用 Windows Installer 技术来打包应用程序，并通过传统安装来部署应用程序。

通过 ClickOnce 部署，可以将应用程序发布到中心位置，然后用户从该位置安装或运行应用程序。通过 Windows Installer 部署，将应用程序打包到 setup.exe 文件中，并将该文件分发给用户，用户可以运行 setup.exe 文件安装应用程序。

18.1.1 什么是 ClickOnce 部署方案

使用 ClickOnce 部署 Windows 应用程序，可以使开发人员在程序发布的时候，相对容易地设置程序的发布方案和更新方案，但是却要求用户在使用 ClickOnce 应用程序时，必须具有相应的网络环境。


所谓的 ClickOnce，英文的意思为“点击一次”，这个名称很形象地说明了使用 ClickOnce 技术部署 Windows 应用程序的特点，即极大地简化了应用程序的安装过程中与用户的交互，当用户确定了要安装指定的 ClickOnce 应用程序时，程序将自动安装并运行。在程序运行期间将根据开发人员对程序指定的更新方法对程序进行更新。

在使用 ClickOnce 部署 Windows 应用程序时，共有 3 种方法发布 Windows 应用程序的安装程序，分别是：

- ☐ 通过网页进行发布；
- ☐ 通过共享文件发布；
- ☐ 通过可移动媒体（如 CD-ROM）发布。

使用 ClickOnce 部署方案部署 Windows 应用程序时，可以通过对 Windows 应用程序发布属性的设置，决定 Windows 应用程序的安装方式。如果需要部署的 Windows 应用程序是用户经常使用的应用程序，可以将其安装在用户的本地计算机上并可在脱机状态下运行。而如果需要部署的 Windows 应用程序的使用率较低，比如只有在每年年底进行业绩总结时使用的应用程序，则没有必要安装在用户的本地计算机上造成资源的浪费，只需要

在用户使用时以联机模式运行指定的应用程序，此时应用程序运行在最终用户的计算机系统缓存中，当对此应用程序使用结束后，操作系统会自动将程序从系统缓存中删除，不在用户的计算机上永久保留任何应用程序的内容。

 **说明：**ClickOnce 应用程序之所以能完成上述的功能，是因为 ClickOnce 应用程序在本质上是隔离的，是完全独立的。


每一个 ClickOnce 应用程序都安装在单独的缓存中，并对此缓存具有独占性，当用户需要使用此应用程序时，ClickOnce 应用程序将从该缓存中启动运行。

如果开发人员希望可以定义应用程序的安装过程，并且希望将应用程序添加到 Windows 系统的注册表中时，则可以使用 Windows Installer 部署方案对 Windows 应用程序进行部署。

18.1.2 什么是 Windows Installer 部署方案

当使用 Windows Installer 部署方案对 Windows 应用程序进行部署时，需要新建一个 Windows 应用程序的安装项目，并向该项目中添加需要安装到最终用户计算机上的文件。使用新建的安装项目并将这些文件打包成安装包，并在 .NET 中设置安装向导外观和步骤。

使用 Windows Installer 部署方案对 Windows 应用程序进行部署，开发人员可以灵活地设置 Windows 应用程序的安装步骤和安装属性，可以通过对安装项目的属性进行设置，制定安装项目的行为特征，比如可以指定应用程序是否可由用户指定安装位置，是否可由计算机的所有使用者使用，是否需要向注册表添加信息，是否在开始菜单和桌面上创建快捷方式。

 **注意：**对于 Windows Installer 应用程序而言，应用系统中的很多组件都是多个应用程序共用的，容易引发使用组件的版本冲突问题，也就是著名的 Dll Hell。

18.2 ClickOnce 部署

在本节中，将通过一个实例详细介绍如何使用 ClickOnce 方案对 Windows 应用程序进行部署。

18.2.1 创建一个部署用的示例程序

在使用 ClickOnce 方案进行部署时，首先要指定需要部署的应用程序，本节中将创建一个简单的示例程序用于发布。

在 Visual Studio 2010 中创建一个新的 Windows 窗体应用程序，将此应用程序命名为 WindowsApplication1，在此程序窗体中添加一个 Label 控件，将此 Label 控件的 Text 属性设置为“ClickOnce 部署”，程序的运行结果如图 18.1 所示。在解决方案的资源管理器中，

可以浏览这个项目中所包含的全部内容,如图 18.2 所示。

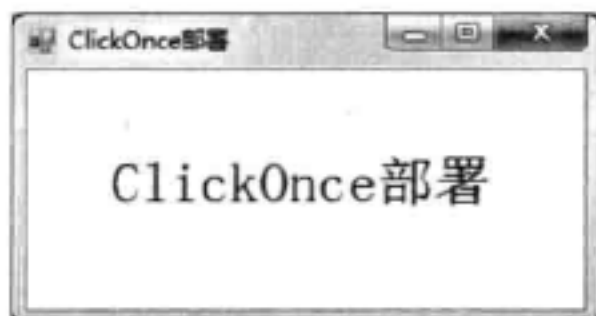


图 18.1 程序运行界面




图 18.2 “解决方案资源管理器”面板

创建好了用于发布的 Windows 应用程序后,对其进行编译并运行。编译通过后,接下来通过启动“发布向导”来对这个应用程序进行发布。

18.2.2 启动“发布向导”的 3 种方法

在.NET 中,一共提供了 3 种方法启动“发布向导”,下面将对这 3 种方法分别阐述。

说明:这 3 种方法的效果相同,读者可根据自己的习惯对这 3 种方法进行选择。

1. 在解决方案资源管理器中打开“发布向导”

- (1) 在“解决方案资源管理器”面板中,选择应用程序项目。
- (2) 右击项目节点,然后选择“发布”命令,将弹出“发布向导”窗体,如图 18.3 所示。



图 18.3 “发布向导”窗口

2. 在“生成”菜单中打开“发布向导”

(1) 在“解决方案资源管理器”中，选择应用程序项目。

(2) 选择“生成”“发布 WindowsApplication1”命令，如图 18.4 所示，将弹出“发布向导”窗体，如图 18.3 所示。



图 18.4 生成菜单

3. 在“项目设计器”中打开“发布向导”

(1) 在“解决方案资源管理器”中，右击应用程序项目并选择“属性”按钮，将弹出“项目设计器”窗体，如图 18.5 所示。



图 18.5 项目设计器

(2) 单击“发布”标签头，在“项目设计器”中将转到“发布”标签页中，然后单击“发布向导”按钮，将弹出“发布向导”窗体，如图 18.3 所示。

当发布向导发布 ClickOnce 应用程序时，可以在此发布向导中对发布的极少数重要属性进行设置，其他属性都取为默认值。

说明：如果需要对发布行为进行特别的说明，则可以在“项目设计器”中的“发布”页中对发布的属性进行设置。

18.2.3 设置发布应用程序的 3 个 URL 地址

在发布 ClickOnce 应用程序时，一共可以对 3 个属性设置相应的 URL 地址，分别是：

- ☐ 发布位置;
- ☐ 安装 URL;
- ☐ 支持 URL。

“发布位置”是开发人员用于设置应用程序将发布到指定的位置，程序发布完成后，程序的安装文件将出现在指定的发布位置。

“安装 URL”是用户下载和安装应用程序的位置，这个位置通常与发布位置相同，此时不需要对此属性进行设置。但是在大型的应用程序开发中，由于权限限制或更新频率要求，开发人员可能将应用程序发布到组织内部的某个共享文件夹中，然后由负责发布的系统管理员将该应用程序移动到安装 URL 指定的位置。此时，发布位置和安装 URL 是两个不同的属性值，需要分别指出。

“支持 URL”用于提供关于应用程序的支持信息，如果对其属性进行设置，则在“WindowsApplication 维护”窗体（参见图 18.24）中的“详细信息”按钮是可用的。单击此按钮，将在浏览器中打开支持 URL 指定的地址。此属性在 ClickOnce 部署中是可选属性。

对于发布位置和安装 URL 的属性值，均可在“项目设计器”的“发布”标签页中进行设置。如图 18.5 所示。

在“项目设计器”的“发布”标签页中单击“选项”按钮，打开“发布选项”对话框，如图 18.6 所示。在此对话框中，向“支持 URL”文本框中输入提供了有关应用程序更多信息的网页路径，设置了 ClickOnce 应用程序的“支持 URL”属性值。



图 18.6 “发布选项”对话框

当应用程序选择的是通过“从 CD-Rom 或 DVD-Rom”运行发布时，那么当用户在本地计算机光驱中插入指定的安装光盘后，可以通过启用 AutoStart 使 ClickOnce 应用程序自动启动。设置此项功能，可在图 18.6 中单击左侧列表中的“部署”项，从显示的选项中选择“对于 CD 安装，插入 CD 时将自动启动安装程序”复选框。

说明：当应用程序选择的是通过“从 CD-Rom 或 DVD-Rom”运行发布，.NET 在发布应用程序时会将一个 Autorun.inf 文件复制到发布位置，使 ClickOnce 应用程序可以自动运行。

18.2.4 为应用程序设置名称

当需要将应用添加到 Windows 系统的开始菜单中时,显示名称与应用程序的程序集名称相同,可以通过设置发布选项中的“产品名称”属性来更改应用程序的显示名称。而对于“开始”菜单中包含应用程序图标文件夹的名称,则可以通过对“发行者名称”属性的设置来实现,具体如图 18.6 所示。

18.2.5 设置需发布的非代码文件

在使用 ClickOnce 发布 Windows 应用程序时,.NET 会默认为程序中所包含的所有非代码文件都一起进行发布。但是,有时候并不希望发布程序中的所有非代码性文件,这时可以在“应用程序文件”对话框中对现有应用程序文件进行设置,以排除不希望或不需发布的文件。

在图 18.5 所示界面中,单击“项目设计器”窗体中的“应用程序文件”按钮,将打开“应用程序文件”对话框,如图 18.7 所示。在“应用程序文件”对话框中,由表格形式列出了程序中所包含的所有非代码性文件的文件名、发布状态和下载组。

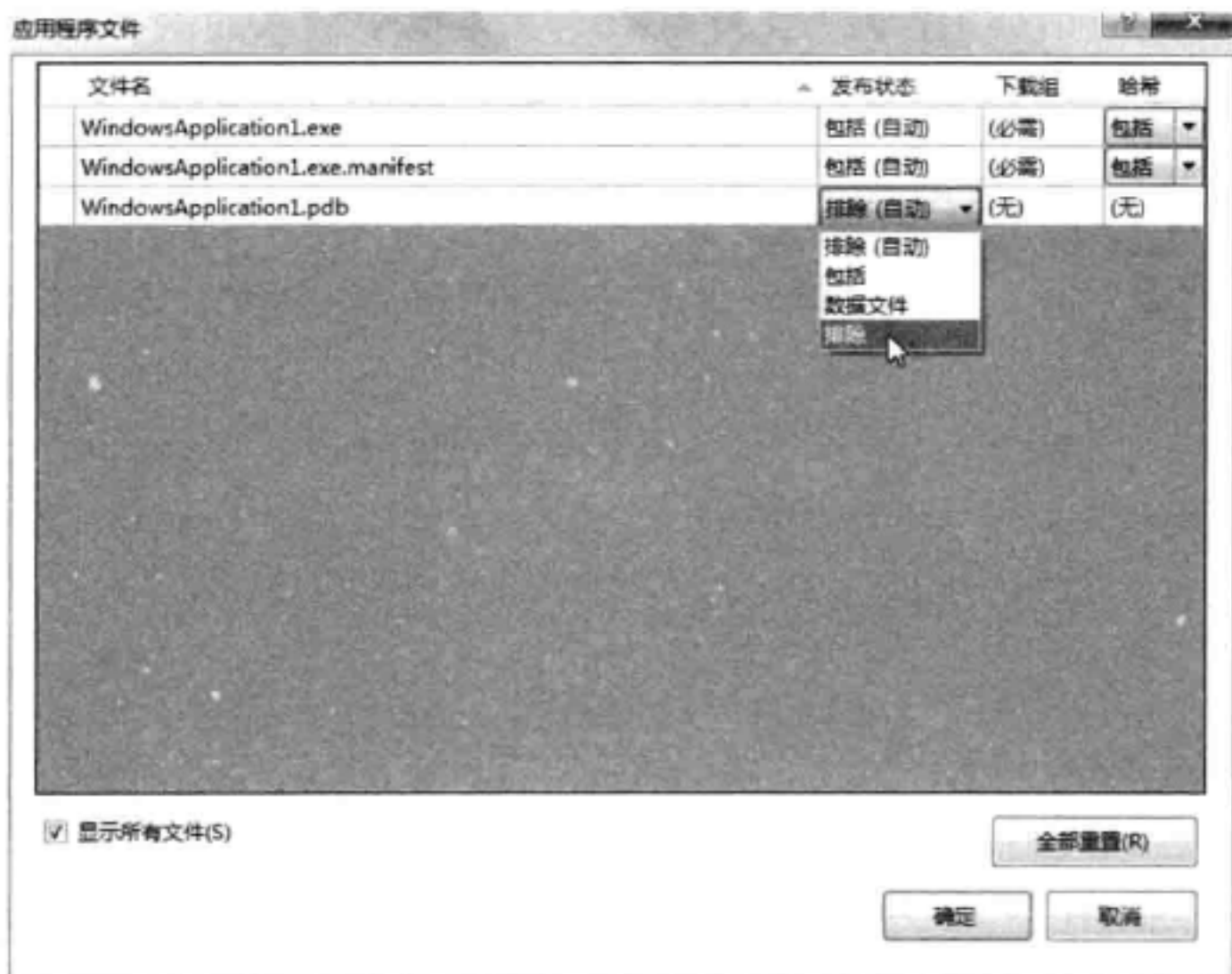


图 18.7 “应用程序文件”对话框 (1)

开发人员可以根据实际的需更改每个文件的发布状态,将此文件标记为数据文件、希望排除的文件或系统必备文件。当需要将文件设置为数据文件时,可以在窗体表格此文件信息行中单击“发布状态”字段旁的下拉按钮,打开下拉列表框,在其中选择数据文件,则此文件被标注为“数据文件”添加到 ClickOnce 应用程序发布包中。

在图 18.7 中,可以观察到在表格中的第一行文件名为 WindowsApplication1.exe 的文件的文件状态被设置为“包括 (自动)”,这是因为在对项目进行发布时,.NET 会将项目中的文件发布状态根据其后缀名设置为默认值,而被程序自动设置的发布状态后面则会添加

“（自动）”两个字。

18.2.6 为应用程序添加下载组

在“应用程序文件”对话框中，除了可以对项目中的非代码性文件的发布状态进行编辑外，还可以对这个文件的下载组进行设置，如图 18.8 所示。

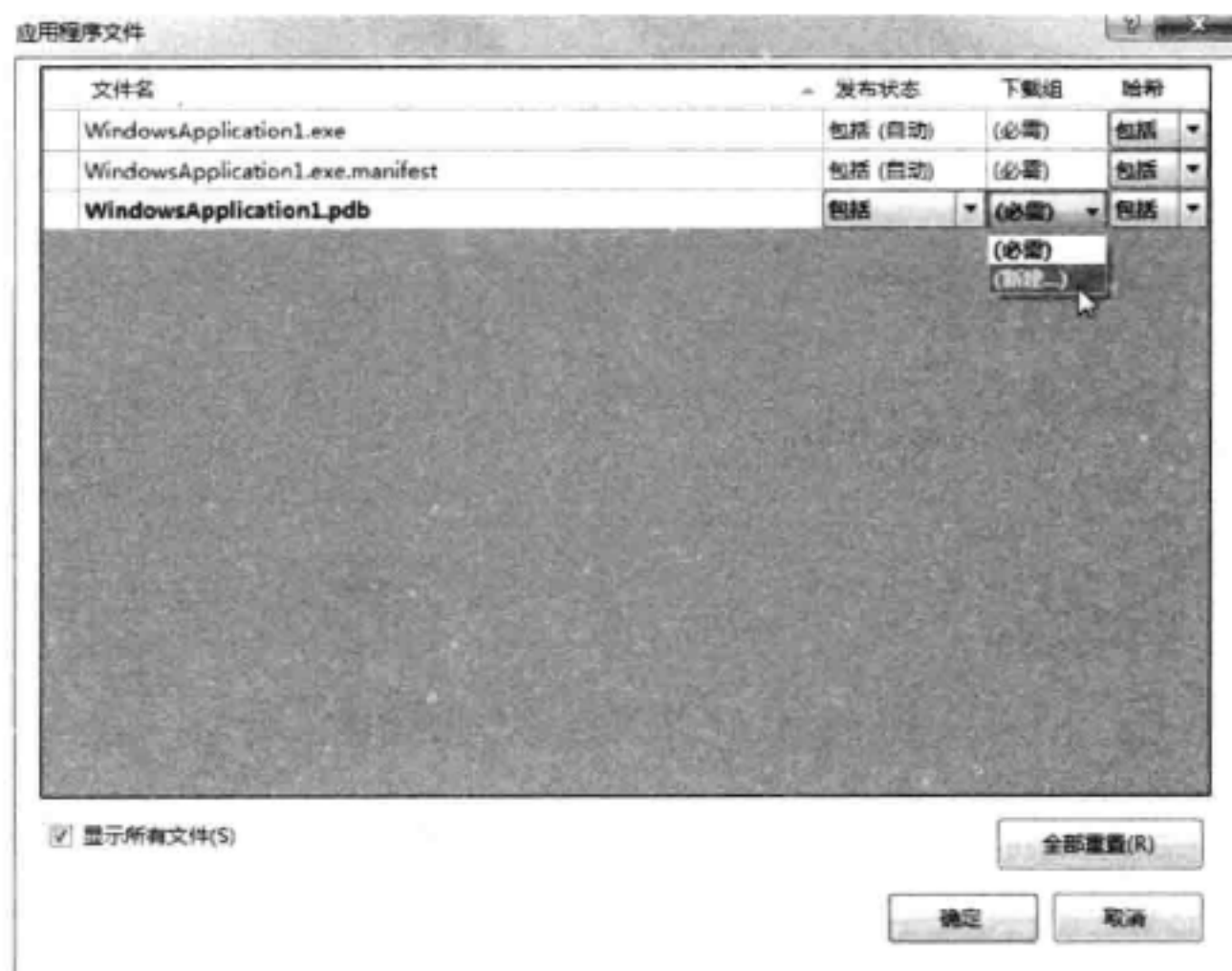


图 18.8 “应用程序文件”对话框（2）

说明：默认情况下，文件的下载组均被设置为“（必需）”。

在初始状态中，所有的应用程序文件都存在于一个默认的下载组“（必需）”中，开发人员可通过创建不同的下载组而使不同的开发文件存在于不同的下载组中。

创建新的下载组的方法是，单击所选文件行中的“下载项”字段旁的下拉按钮，打开下拉列表框，在其中选择“（新建...）”，则弹出如图 18.9 所示的“新建组”对话框。

在“新建组”对话框中，输入新建组的名称，然后单击“确定”按钮。



图 18.9 “新建组”对话框

则一个新的组名出现在“下载组”字段的下拉列表中。开发人员可以通过在下拉列表中选择文件所在处的新的下载组，为不同的下载文件添加新的下载组。

说明：对不同的下载文件设置了不同的下载组后，ClickOnce 应用程序将在启动时，根据所检测到的最终用户的系统情况而对文件进行分组安装。

18.2.7 设置应用程序的系统必备组件

运行所有的.NET 开发的应用程序，必须保证在最终用户的本地系统中安装有正确版本

的.NET Framework。如果目标系统中并没有安装程序运行的必备组件,则可以通过“系统必备”对话框(如图18.10所示)来添加系统必备的组件与应用程序一起打包安装。



图 18.10 “系统必备”对话框

在安装 ClickOnce 应用程序时,安装程序会自动检测目标系统中是否安装了指定的系统必备组件,如果系统已存在这些组件,则会继续进行应用程序的安装;如果目标系统中没有安装这些组件,则安装程序会先安装指定的组件然后再继续安装应用程序。

在“项目设计器”对话框中,进入“发布”标签页(如图18.5所示),单击标签页上的“系统必备”按钮,将弹出“系统必备”对话框。

在此对话框中,选中“创建用于安装系统必备组件的安装程序”复选框。注意,此复选框必须为选中状态。在下面的列表窗体中选择应用程序的必备组件。如果希望安装包装不打包系统必备组件,则可以指定必备组件的下载位置。当目标系统中不包含指定组件时,安装程序会自动到指定的网址对组件进行下载。可以在“指定系统必备组件的安装位置”单选区域中,选择“从下列位置下载系统必备组件”单选按钮,然后从下拉列表中选择或输入一个 URL,单击“确定”按钮后完成安装程序“系统必备”组件的设置。

在完成了 ClickOnce 应用程序的相应 URL 设置、必备组件设置和备份文件的设置后,需要 ClickOnce 应用程序的访问权限和安全性进行设置。

18.2.8 设置安装组件的权限

在通常情况下,开发人员并不需要对 ClickOnce 应用程序的访问权限进行设置,.NET 会自动计算并设置出符合应用程序需求的访问权限。但是当 ClickOnce 应用程序的权限设置不合理时,应用程序的部署和安装是不能完全进行的,安装程序会弹出提示信息,指出程序的访问权限不够,程序安装无法继续进行,并回滚安装程序。

可以在“项目设计器”对话框的“安全性”标签页中,对程序中组件的访问权限进行设置,如图18.11所示。选中该标签页中的“启用 ClickOnce 安全设置”复选框,然后在下面的列表中对此程序的安全性进行具体设置。

在进行权限设置时应为应用程序配置符合实际需要的访问安全权限。权限设置过小,

程序将无法安全进行，权限设置过大，则会引起不必要的安全隐患。



图 18.11 ClickOnce 安全设置

说明：在 .NET 中，提供了完备的权限分析工具来确定应用程序所需要的安全访问权限，开发人员可借助这些工具来对程序的访问规则进行设置。

18.2.9 应用程序的 3 种发布方式

在对 Windows 应用程序的发布属性进行设置后，就可以正式开始进行 Windows 应用程序的发布，对于使用 ClickOnce 方案部署 Windows 应用程序，一共有 3 种不同的策略可以选择。这 3 种部署策略分别为：

- ☐ 发布到网站；
- ☐ 发布到共享文件夹；
- ☐ 发布到 CD-ROM。

在部署 ClickOnce 应用程序时，所选择的策略主要取决于要部署的应用程序的类型。下面将通过具体的发布步骤，来对 3 种发布策略进行说明。

18.2.10 将应用程序发布到网站

使用此部署策略时，应用程序的安装文件将被部署到指定的网站服务器上，最终用户需要登录网站通过双击此应用程序的安装文件图标进行安装。

注意：这种部署方案要求客户端必须保持畅通的网络连接。对于客户端具有高速网络连接的应用程序，比较适合使用这种发布策略。

将 Windows 应用程序发布到网站可以经过以下几个步骤：

- (1) 使用任意方式打开“发布向导”，如图 18.3 所示。
- (2) 在“要在何处发布该应用程序？”对话框中，输入格式为“http://www.microsoft.com/”

foldername”的一个有效的 URL，然后单击“下一步”按钮。

(3) 在“应用程序是否将脱机使用?”对话框中，如图 18.12 所示，根据需要选择适当的选项，如果此应用程序是最终用户的常用程序，那么此应用程序在脱机状态下也应当可以正常运行，则单击“是，该应用程序可以联机或脱机使用”按钮，这种情况下，当安装完成后，程序的快捷方式将出现在目标系统的“开始”菜单中；如果此应用程序是使用频率较低的应用程序，且用户可以保持较为高速稳定的网络连接，则此应用程序每次从发布位置直接启动，不占用客户端的物力资源，则单击“否，该应用程序只能联机使用”按钮。那么目标系统将不会对此应用程序进行物理安装，系统的“开始”菜单中也不会创建此应用程序的快捷方式。单击“下一步”按钮继续。

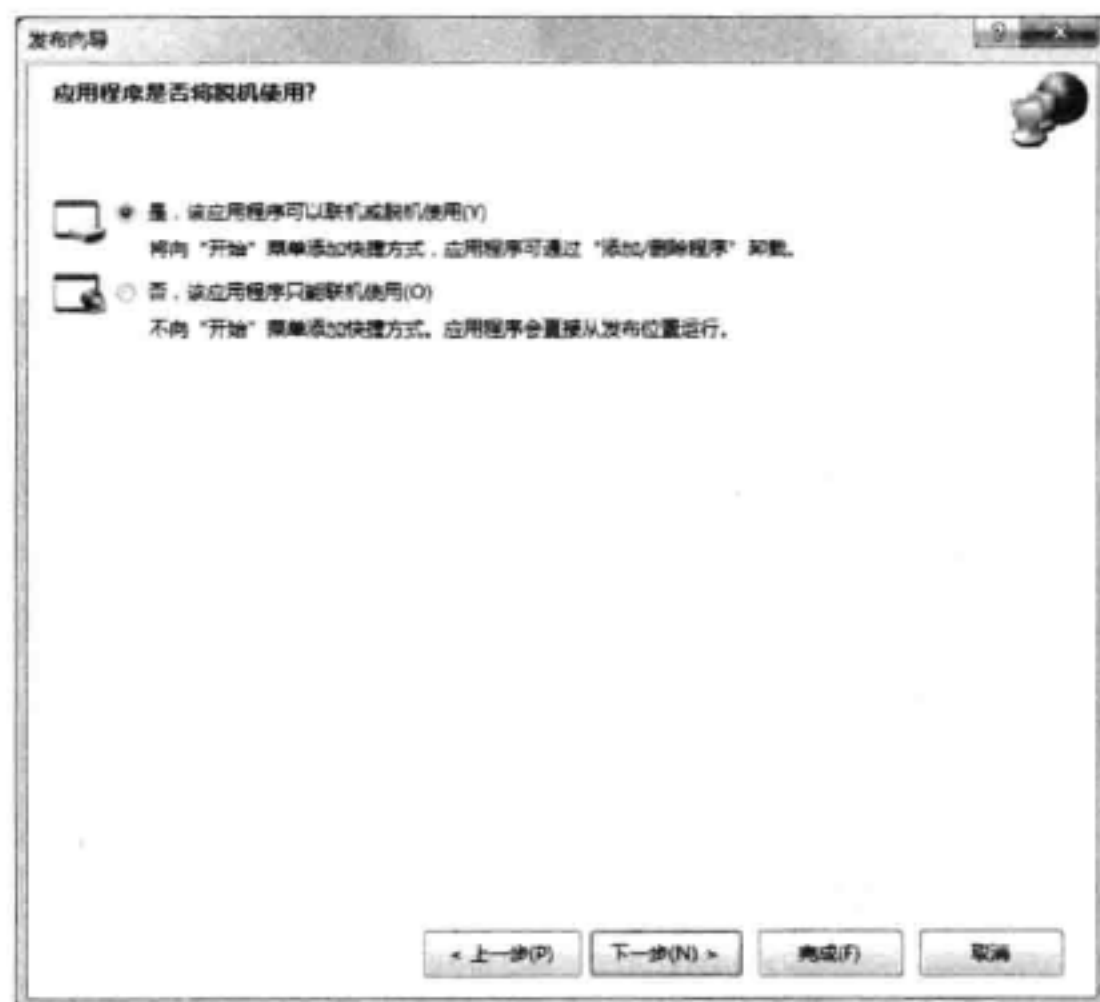


图 18.12 “应用程序是否将脱机使用?”对话框


(4) 发布向导进入“发布准备就绪”对话框，如图 18.13 所示。在图中，将给出此次发布的具体信息，主要是开发人员所做的选择和对此次发布的属性设置的相关信息，单击“完成”按钮以发布应用程序。



图 18.13 “发布准备就绪!”对话框

18.2.11 将应用程序发布到共享文件夹

使用此部署策略时，应用程序的安装文件将被部署到指定的共享文件上，而最终用户却可以通过 3 种方式进行下载安装。

 **注意：**当客户端全部位于某局域网内时，可指定应用程序通过共享文件夹进行安装。

将 Windows 应用程序发布到共享文件夹可以经过以下几个步骤：

(1) 使用任意方式打开“发布向导”对话框，如图 18.3 所示。

(2) 在“要在何处发布该应用程序？”页面中，输入格式为“\\pc\application”的一个有效的文件路径，然后单击“下一步”按钮。

(3) 在“用户如何安装应用程序？”对话框中，如图 18.14 所示，开发人员可以选择用户下载应用程序安装包的位置，此页面只有在“要在何处发布该应用程序？”页面中的“指定发布此应用程序的位置”的文本框中输入的不是标准的 URL 格式的文本时，才会出现。因为当此文本框中输入的是标准的 URL 格式的文本信息时，安装程序会默认为应用程序的发布地址和应用程序的下载更新地址是一个地址，因此开发人员无需再指定新的程序下载 URL。



图 18.14 “用户如何安装应用程序？”对话框

当将程序发布到目的地后，如果希望用户从网站安装应用程序，则选中“从网站”单选按钮，并输入希望用户下载安装文件的有效的 URL，然后单击“下一步”按钮。如果用户从共享文件夹中安装应用程序，则选中“从 UNC 路径或文件共享”单选按钮，然后单击“下一步”按钮。


(4) 在“应用程序是否将脱机使用？”页面中，如图 18.12 所示，根据需要进行适当的

选项, 如果此应用程序是最终用户的常用程序, 那么此应用程序在脱机状态下也应当可以正常运行, 则单击“是, 该应用程序可以联机或脱机使用”按钮, 这种情况下, 当安装完成后, 程序的快捷方式将出现在目标系统的“开始”菜单中; 如果此应用程序是使用频率较低的应用程序, 且用户可以保持较为高速稳定的网络连接, 则此应用程序每次从发布位置直接启动, 不占用客户端的物力资源, 则单击“否, 该应用程序只能联机使用”按钮, 那么目标系统将不会对此应用程序进行物理安装, 系统的“开始”菜单中也不会创建此应用程序的快捷方式。单击“下一步”按钮继续安装。

(5) 发布向导进入“发布准备就绪”页面, 如图 18.13 所示, 在图中, 将给出此次发布的具体信息, 主要是开发人员所作的选择和对此次发布的属性设置的相关信息, 单击“完成”按钮以发布应用程序。

18.2.12 将应用程序发布到 CD-ROM

使用此部署策略时, 应用程序的安装文件将被部署到 CD-ROM 上。

 **注意:** 如果以上两个方法都不适用时, 可通过 CD-ROM 进行安装。

将 Windows 应用程序发布到 CD-ROM 可以经过以下几个步骤。

(1) 使用任意方式打开“发布向导”, 如图 18.3 所示。

(2) 在“要在何处发布该应用程序?”页面中, 输入格式为“c:\application”的一个有效的文件路径, 然后单击“下一步”按钮。

(3) 在“用户如何安装应用程序?”页面中, 如图 18.14 所示, 选中“从 CD-ROM 或 DVD-ROM”单选按钮, 然后单击“下一步”按钮。

(4) 在“应用程序是否将脱机使用?”页面中, 如图 18.12 所示, 根据需要进行适当的选项, 如果此应用程序是最终用户的常用程序, 那么此应用程序在脱机状态下也应当可以正常运行, 则单击“是, 该应用程序可以联机或脱机使用”按钮, 这种情况下, 当安装完成后, 程序的快捷方式将出现在目标系统的“开始”菜单中; 如果此应用程序是使用频率较低的应用程序, 且用户可以保持较为高速稳定的网络连接, 则此应用程序每次从发布位置直接启动, 不占用客户端的物力资源, 则单击“否, 该应用程序只能联机使用”按钮, 那么目标系统将不会对此应用程序进行物理安装, 系统的“开始”菜单中也不会创建此应用程序的快捷方式。单击“下一步”按钮继续安装。

(5) 当应用程序是通过 CD-ROM 安装的, 那么当程序需要自动更新时, 可以在安装程序中指定检查和下载更新的网络位置。在“应用程序将到哪里检查更新?”页面中, 如图 18.15 所示, 能够对程序是否可以更新进行设置。如果应用程序能够检查是否有可用的更新, 则选中“该应用程序将从下列位置检查更新”单选按钮, 并在相应文本框中输入检查可用更新的 URL; 如果应用程序不需要检查更新, 则选中“该应用程序将不检查更新”单选按钮, 单击“下一步”按钮继续安装。

(6) 发布向导进入“发布准备就绪”页, 如图 18.13 所示。在图中, 将给出此次发布的具体信息, 主要是开发人员所做的选择和对此次发布的属性设置的相关信息, 单击“完成”按钮以发布应用程序。



图 18.15 “应用程序将到哪里检查更新？”对话框

18.2.13 发布应用程序

在 18.2.12 节中，选择了发布策略后，单击“完成”按钮发布应用程序，此时发布状态显示在任务栏的状态通知区域中，用于显示发布进行的状态，如图 18.16 所示。



图 18.16 发布状态

程序发布完成后，打开在发布向导中指定的应用程序发布到的文件夹。可以看到在此文件夹中，共有 3 个文件，如图 18.17 所示：

- ❑ 一个名为 setup.exe 的应用程序，可以用来安装应用程序。
- ❑ 一个文件夹 Application Files，包含有应用程序的内容及部署信息。在该文件夹中又有一个名为 WindowsApplication1_1_0_0_0 的文件夹，里面包含 4 个文件。
- ❑ 一个 ClickOnce 应用程序部署清单，在图 18.17 中，为 WindowsApplication1.application。



图 18.17 发布状态

用记事本打开 ClickOnce 应用程序部署清单文件 WindowsApplication1.application，可看到如图 18.18 所示的内容。此清单用于描述如何部署应用程序，包含的主要内容有：



图 18.18 部署清单


- ☐ 应用程序清单的位置；
- ☐ 客户端应运行的应用程序的版本。

用记事本打开位于 Application Files \WindowsApplication1_1_0_0_0 文件夹中的应用程序清单文件 WindowsApplication1.exe.manifest，可看到如图 18.19 所示的内容。此清单用于描述应用程序本身，包含的主要内容有：



图 18.19 应用程序清单

- ☐ 应用程序中所包含的程序集；
- ☐ 组成应用程序的依赖项和文件；
- ☐ 应用程序所需的权限；
- ☐ 应用程序的更新位置。

 **说明：**当部署清单和应用程序清单生成后，开发人员或发布人员需要将新的应用程序清单和部署清单复制到指定的应用程序发布位置，以及用户下载更新的网站服务器或共享文件夹中，最终用户可以通过单击部署清单的图标对程序进行下载和更新。

18.2.14 安装发布的应用程序

ClickOnce 应用程序部署结束后，最终用户可以登录相应的网站、共享文件夹或从 CD-ROM 中对文件进行安装，单击应用程序的部署清单的文件图标，程序将弹出“正在启动应用程序”消息框，如图 18.20 所示，指示启动或安装应用程序所处的状态。

对应用程序要求验证完成后，安装程序将自动弹出如图 18.21 所示的对话框，询问用户是否确认要安装此应用程序。在 ClickOnce 应用程序中，这也是唯一需要用户执行的操作。

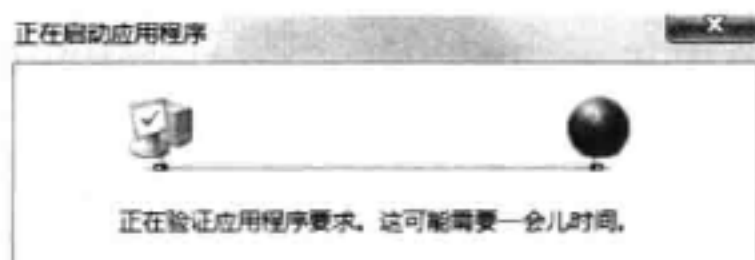


图 18.20 “正在启动应用程序”消息框

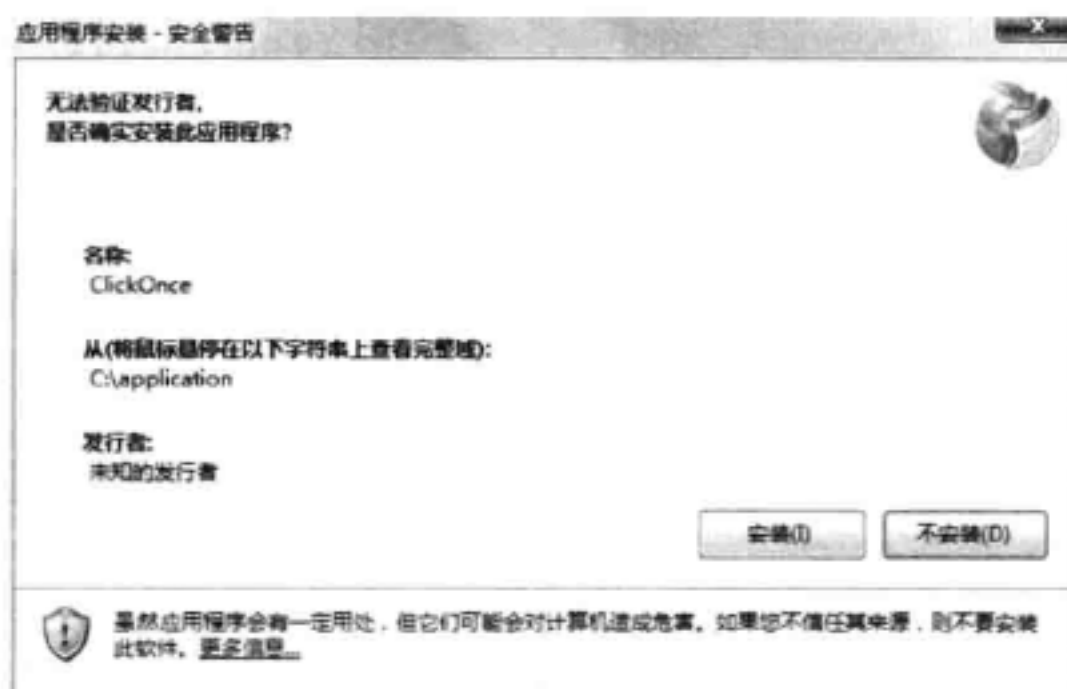



图 18.21 “应用程序安装”对话框

 **说明：**当无法验证程序的安全性时，最终用户可通过单击对话框下方的“更多信息”文本标签来打开如图 18.22 所示的提示信息。

当用户在图 18.21 所示对话框中单击“安装”按钮后，将弹出程序安装进度对话框，如图 18.23 所示。在程序安装完成后，此对话框自动关闭，用户也可以通过单击“取消”按钮来回滚已执行的安装。

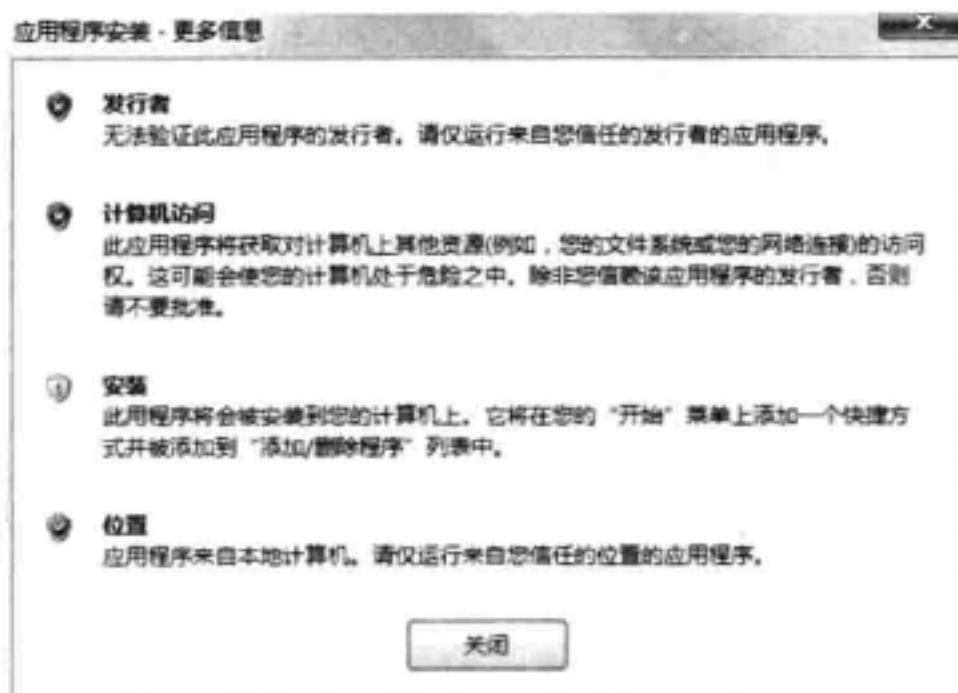


图 18.22 更多信息图



18.23 安装进度对话框

安装完成后，在“开始”菜单中就可看到安装应用程序的快捷方式，通过该快捷方式

即可启动应用程序。如果要对此应用程序进行修复或删除,可通过“控制面板”中的“卸载或更改程序”来完成。在“控制面板”的“卸载或更改程序”列表中找到安装的“ClickOnce”程序,双击后打开应用程序维护窗体,如图 18.24 所示,最终用户可以通过此窗体删除 ClickOnce 应用程序。在图 18.24 中单击“详细信息”按钮将打开发布应用程序时设置的“支持 URL”所设置的网址。

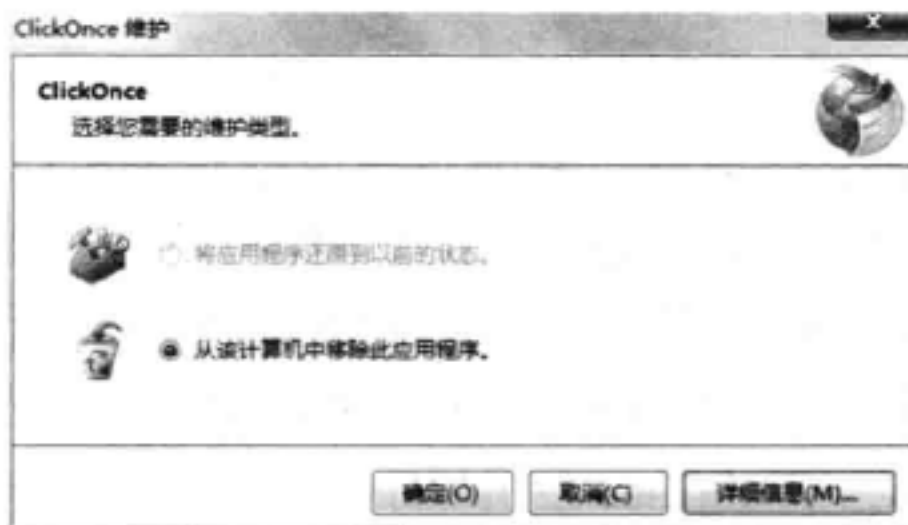


图 18.24 应用程序维护窗体

18.3 Windows Installer 部署

在本节中,将通过一个具体的例子详述如何使用 Windows Installer 方案对 Windows 应用程序进行部署。

18.3.1 创建一个部署用的示例程序

在使用 Windows Installer 方案进行部署时,首先要指定需要部署的应用程序,在本节中将创建一个简单的示例程序用于创建安装包。

使用 Visual Studio 2010 创建一个新的 Windows 窗体应用程序,将此应用程序命名为 WindowsApplication2,在此程序窗体中添加一个 Label 控件,并将该控件的 Text 属性设置为“Windows Installer 部署”,程序的运行结果如图 18.25 所示。在解决方案的资源管理器中,可以浏览到这个项目中所包含的全部内容,如图 18.26 所示。

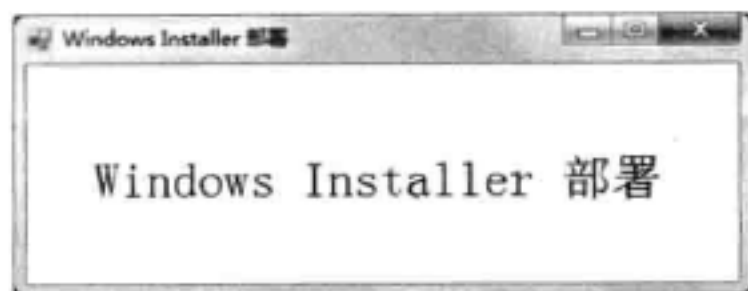


图 18.25 程序运行界面



图 18.26 解决方案资源管理器

创建好用于发布的 Windows 应用程序后,对其进行编译并运行。编译通过后,通过创建“安装和部署项目”来新建这个应用程序的安装包。

18.3.2 创建部署项目

在 Windows 应用程序创建完成后,则需要向此应用程序添加“安装和部署项目”。

(1) 在“解决方案资源管理器”中选中“解决方案”节点,单击此节点的快捷菜单中的“添加|新建项目”命令,打开“添加新项目”对话框,如图 18.27 所示。



图 18.27 “添加新项目”对话框


(2) 在“添加新项目”对话框中的“项目类型”树结构中,选择“其他项目类型”节点的“安装和部署项目”子节点,接着选择 Visual Studio Installer 选项,然后在对话框右侧的“模板”视图选择“安装项目”选项,单击“确定”按钮关闭对话框。这时新增的项目被添加到“解决方案资源管理器”面板中,如图 18.28 所示。



图 18.28 “解决方案资源管理器”面板

18.3.3 设置文件安装到目标机器的位置

“安装和部署项目”被添加后,Visual Studio 的窗体设计器中将自动打开“文件系统编辑器”,如图 18.30 所示。

 **说明：**也可以通过选择菜单上的“视图”|“编辑器”|“文件系统”命令打开“文件系统编辑器”，如图 18.29 所示。

“文件系统”编辑器以虚拟的方式模拟出目标机器的文件系统，开发人员可根据希望应用程序安装的位置在“文件系统”编辑器中进行配置。如图 18.30 所示“应用程序文件夹”相当于目标系统的 Program Files 文件夹，“桌面文件夹”相当于目标系统的“桌面”，“用户的‘程序’菜单”相当于目标系统的“开始”菜单。



图 18.29 菜单项

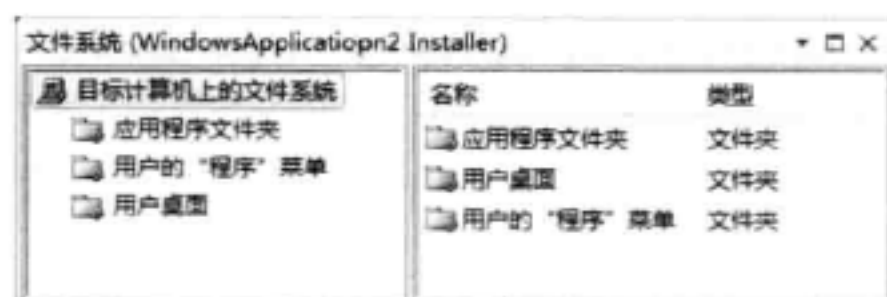


图 18.30 文件系统

开发人员还可以在“文件系统”编辑器的虚拟文件夹中创建新的文件夹，并将需要部署的应用程序的文件添加到新增的文件夹中，此文件将在安装文件时自动添加到各虚拟文件夹所对应的文件夹中。

“文件系统”编辑器中的虚拟文件夹中包含着应用程序运行所需要的各种信息，主要包括：

- ☐ 文件；
- ☐ 项目的输出项；
- ☐ 程序集。

项目输出中的项目是指在解决方案中的安装和部署项目以外的项（在本例中，是 WindowsApplication2），项目输出项的主要内容有：

- ☐ 主输出：包含由项目生成的.dll 文件和.exe 文件；
- ☐ 本地化资源：包含每种区域性资源的附属程序集；
- ☐ 调试符号：包含对项目进行调试时生成的调试文件；
- ☐ 内容文件：包含项目中的所有内容文件；
- ☐ 源文件：包含项目中的所有源文件；
- ☐ 文档文件：包含项目中的 XML 文档文件；
- ☐ XML 序列化程序集：包含项目中的 XML 序列化程序集。

18.3.4 添加项目输出组

了解了“文件系统”编辑器后，可以使用“文件系统”编辑器向安装项目中添加 Windows 应用程序及其他文件。将基于 Windows 的应用程序添加到安装程序中，通常可以经过以下几个步骤：

- (1) 在“解决方案资源管理器”中选中“WindowsApplication2 Installer”项目。
- (2) 在相应的“文件系统”编辑器中，选择“应用程序文件夹”节点，右击鼠标打开

此节点的快捷菜单。

(3) 在快捷菜单上指向“添加”命令，然后单击“项目输出”按钮，弹出如图 18.31 所示的“添加项目输出组”对话框。

(4) 在“添加项目输出组”对话框中，从“项目”下拉列表中选择 WindowsApplication2，当应用程序中只有一个非安装和部署项目时，则此项目为默认选中项目。

(5) 在“添加项目输出组”对话框下面的列表视图选择“主输出”，并在“配置”下拉列表框中，将其设置为“(活动)”状态，单击“确定”按钮后关闭对话框。

注意：此时在“文件系统”编辑器的“应用程序文件夹”节点所对应的右侧视图中，将添加 WindowsApplication2 项目的主输出，如图 18.32 所示。



图 18.31 “添加项目输出组”对话框



图 18.32 “应用程序文件夹”内容

18.3.5 设置安装后的桌面快捷方式

通常情况下，在安装了应用程序后，用户都希望安装程序可以向桌面自动添加一个应用程序的快捷方式。向桌面添加 Windows 应用程序的快捷方式可以经过以下几个步骤：

(1) 在“解决方案资源管理器”中选中 WindowsApplication2 Installer 项目。

(2) 在相应的“文件系统”编辑器中，选择“应用程序文件夹”节点，“文件系统”编辑器的右侧视图中选择“主输出来自 WindowsApplication2 (活动)”节点，单击鼠标右键弹出快捷菜单。

(3) 在快捷菜单上单击“创建 主输出来自 WindowsApplication2 (活动) 的快捷方式”，如图 18.33 所示，将会在右侧窗体中添加一个名为“主输出来自 WindowsApplication2 (活动) 的快捷方式”节点。

(4) 将该快捷方式重命名为“WindowsApplication2”。

(5) 选择创建的快捷方式“WindowsApplication2”，将它拖到左窗格的“用户桌面”文件夹中。

此时在“文件系统编辑器”的“用户桌面”节点所对应的右侧视图如图 18.34 所示。

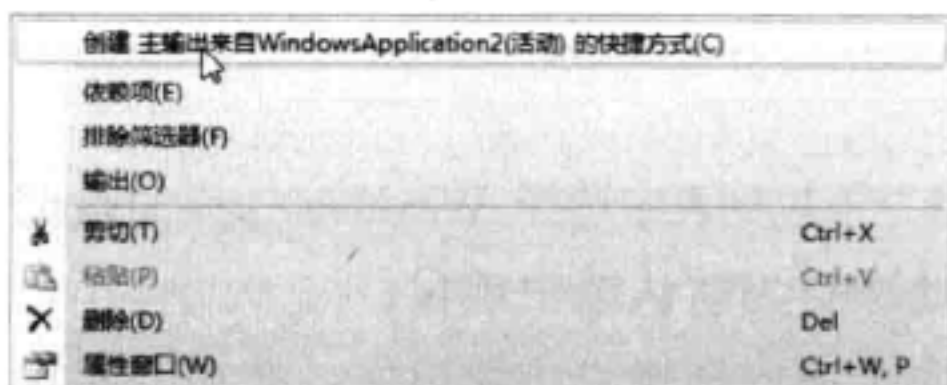


图 18.33 添加应用程序的快捷方式

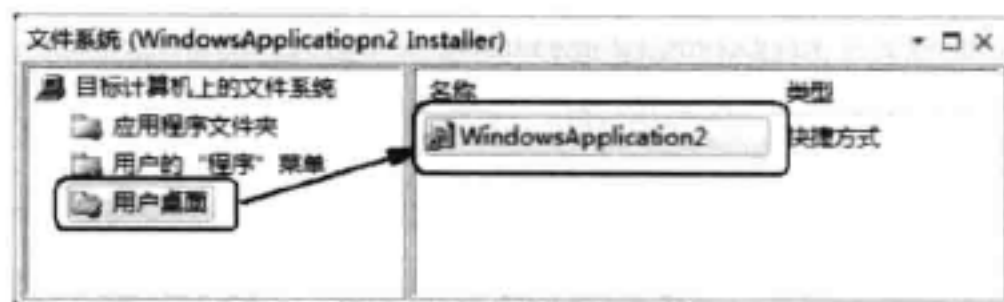


图 18.34 “用户桌面”内容

18.3.6 更改安装应用程序的部署特性

在“文件系统”编辑器中添加了安装程序所需要发布的文件后，可以对安装程序的属性进行设置。安装和部署项目的属性列表如图 18.35 所示。



图 18.35 属性列表

说明：通过对这些属性进行设置，可以更改安装的应用程序的部署特性。

下面将就一些常用的属性进行说明。

InstallAllUsers 属性是一个 bool 类型的属性值，用于指示当前的应用程序是仅为安装用户使用，还是所有使用计算机的用户都可使用。

Manufacturer 属性是一个字符串形式的属性值，用于指示应用程序或组件的制造商名称，这个名称也会作为安装后的应用程序所在文件夹的名称。

ProductName 属性是一个字符串形式的属性值，用于指示应用程序和组件的名称。

Title 属性是一个字符串形式的属性值，用于指示安装程序的标题，此属性所定义的字符串将出现在安装对话框的标题栏处。

18.3.7 将应用程序信息添加到注册表

在对 Windows 应用程序的属性进行设置后,开发人员可以通过“注册表编辑器”将应用程序的有关信息添加到注册表中。将 Windows 应用程序的相关信息添加到注册表中可以经过以下几个步骤:

(1) 在“解决方案资源管理器”中选中 WindowsApplication2 Installer 项目。在“视图”菜单上指向“编辑器”选项,然后选择“注册表”项目,如图 18.29 所示。此时,将会显示“注册表编辑器”窗口,如图 18.36 所示。

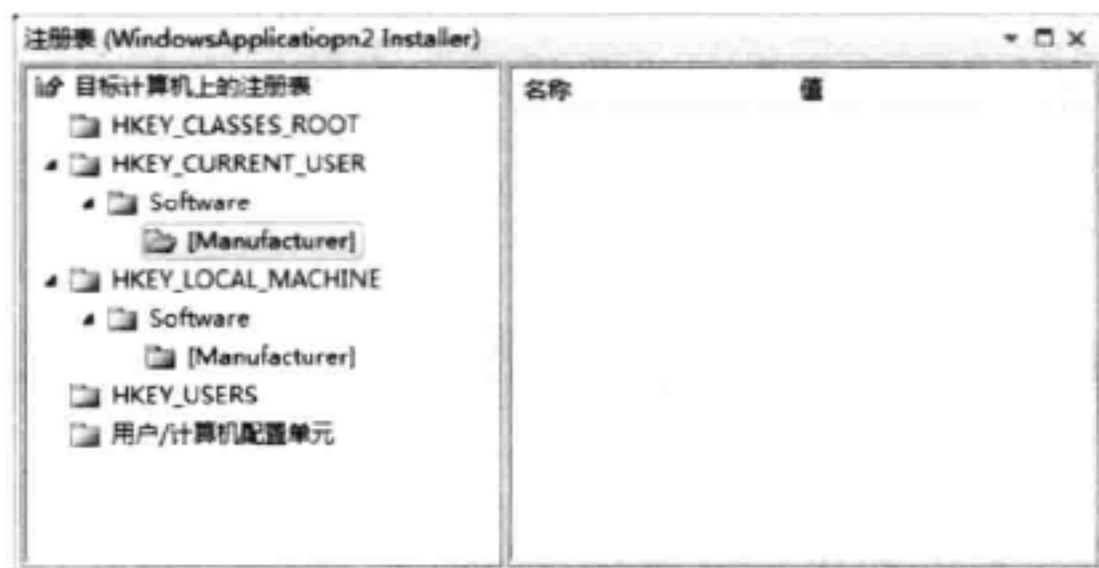


图 18.36 发布状态

(2) 在相应的“注册表编辑器”中,选择“HKEY_CURRENT_USER | Software | [Manufacturer]”节点并展开它。[Manufacturer]节点用方括号括起来,表示它是一个属性,将被替换为输入的部署项目的 Manufacturer 属性值。

(3) 在快捷菜单上选择“新建”|“键”命令,在选定的节点下将自动添加一个新的注册表项,并将其重命名为 ApplicationSetting,选定它并在快捷菜单上选择“新建”|“字符串值”命令,将字符串值重命名为 FormColor。

(4) 选中新建的字符串项,在“属性”窗口中,将 Value 属性值设置为 Red。

18.3.8 安装过程中可使用的对话框

设置好安装程序所包含的文件,创建相应的快捷方式,设置相关属性和注册表项后,就可以对安装程序所涉及的对话框进行统一的设置,以控制安装程序执行时,用户所执行的操作。

在安装程序中,开发人员可定义和增/删的对话框有以下几种:

- ☐ “欢迎使用”用户界面对话框;
- ☐ “安装文件夹”用户界面对话框;
- ☐ “确认安装”用户界面对话框;
- ☐ “进度”用户界面对话框;
- ☐ “已完成”用户界面对话框;
- ☐ “单选按钮”用户界面对话框;
- ☐ “复选框”用户界面对话框;
- ☐ “客户信息”用户界面对话框;

- ❑ “文本框”用户界面对话框；
- ❑ “许可协议”用户界面对话框；
- ❑ “自述文件”用户界面对话框；
- ❑ “注册用户”用户界面对话框。

在“解决方案资源管理器”中选择“WindowsApplication2 Installer”项目。在“视图”菜单上指向“编辑器”命令，然后选择“用户界面”选项，此时，将会显示“用户界面编辑器”窗口，如图 18.37 所示。

在图 18.37 中，列举了安装程序过程中常用的界面。Visual Studio 2010 将这些安装程序界面分为了“普通用户安装”界面和“管理员安装”界面两种。每一种中又分为了“启动”、“进度”和“结束”3 个类别。

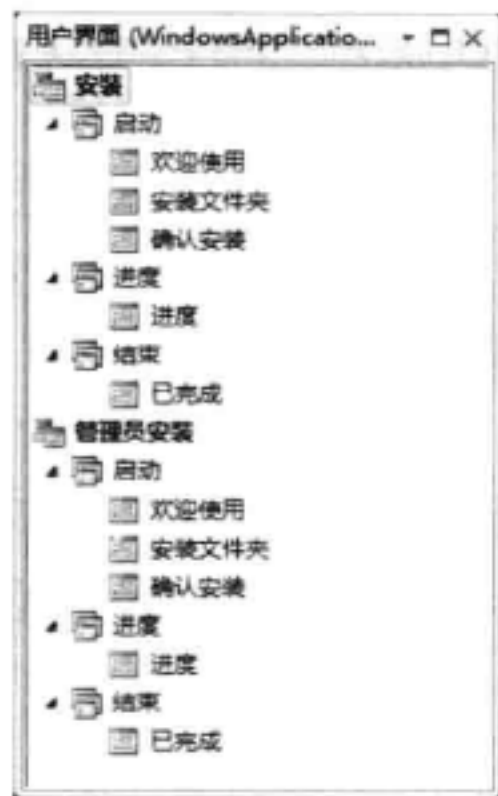


图 18.37 用户界面编辑器

说明：开发人员可通过在“启动”界面类别中增删对话框来使用户灵活地定义应用程序的安装属性。

下面，将对默认的安装过程对话框进行说明。

18.3.9 “欢迎使用”用户界面对话框

“欢迎使用”对话框的最终界面如图 18.38 所示。此界面主要用于介绍此安装向导的相关信息 and 应用程序的版权说明。



图 18.38 “欢迎使用 WindowsApplication2 安装向导”对话框

在对话框的顶部标题处写有“欢迎使用 WindowsApplication2 安装向导”。图 18.38 中的其他文本内容和图示可在“欢迎使用”用户界面对话框的属性窗体中进行设置。

说明：WindowsApplication2 是在前面部署项目的属性时，对 ProductName 属性所设置的属性值。

“欢迎使用”用户界面对话框的 WelcomeText 属性用来设置在对话框正中的正文文

本，默认值为“安装程序将引导您完成在您的计算机上安装 WindowsApplication2 所需的步骤”，开发人员也可将此属性值设置为其他用于说明此安装向导的文字。

而 CopyrightWarning 属性用来设置在对话框底部的版权说明文本，默认值为“警告：本计算机程序受版权法和国际条约保护。如未经授权而擅自复制或传播本程序（或其中任何部分），将受到严厉的民事及刑事制裁，并将在法律许可的范围内受到最大程度的起诉”，开发人员也可将此属性值设置为其他用于说明版权信息的文字。

另外，开发人员还可以通过设置 BannerBitmap 属性来设置对话框中标题区右侧的图像信息，默认图像信息如图 18.38 所示。


18.3.10 “选择安装文件夹”用户界面对话框

“选择安装文件夹”对话框的最终界面如图 18.39 所示，在此对话框中用户可以通过单击“浏览”按钮选择应用程序所安装的文件夹。“文件夹”字段的默认值由安装和部署项目的 DefaultLocation 属性确定。如果用户对此文件夹进行改写，也会改变安装程序相应的 DefaultLocation 属性。



图 18.39 “选择安装文件夹”对话框

将对话框的 InstallAllUsersVisible 属性设置为 true 时，则在对话框下方会增加一个新的控件组，其中包括一个内容为“为自己还是为所有使用该计算机的人安装 WindowsApplication2”的文本标签，以及两个单选按钮。单选按钮的文本分别是“任何人”和“只有我”。

说明：用户在安装过程中通过对单选按钮的选择可以自定义应用程序是否可为该计算机的所有用户所使用。

18.3.11 “确认安装”用户界面对话框

“确认安装”对话框的最终界面如图 18.40 所示，此对话框是安装程序启动部分的最后一个界面，用于询问用户是否已经确认了前面对应用程序所做的各种设置，也是在启动阶段，取消程序安装的最后一次机会。



图 18.40 “确认安装”对话框

如果用户对先前的设置不够满意，则可以单击“上一步”按钮回到前面的对话框进行更改，一旦单击了“确定”按钮，则所做的设置将不能更改。用户也可以单击“取消”按钮取消本次安装。

对于此对话框，开发人员同样可以通过更改 `BannerBitmap` 属性，来更改对话框标题栏右侧的图标。

18.3.12 “进度”用户界面对话框

完成了安装过程中“启动”部分的对话框设置，则进入“进度”部分的对话框设置。在“进度”部分中，Visual Studio 默认的对话框只有“进度”对话框一个，如图 18.41 所示。



图 18.41 “进度”对话框

“进度”对话框主要通过一个进度条控件，在程序的安装过程中向用户显示当前程序的安装进度。开发人员也可以通过设置此对话框的 `ShowProgressBar` 属性隐藏进度条控件。

18.3.13 “安装完成”用户界面对话框

应用程序安装完成后，将启动“安装完成”对话框，如图 18.42 所示，此对话框用于

通知用户应用程序的安装已完成。



图 18.42 “安装完成”对话框

说明：“安装完成”对话框始终是安装过程中显示的最后一个对话框，因此只能位于“用户界面编辑器”的“结束”部分。

开发人员可以通过更改“安装完成”用户界面对话框的 UpdateText 属性值来更改对话框中所显示的文本内容，此属性的默认值为“请使用 Windows Update 检查是否有重要的 .NET Framework 更新”。同样，开发人员也可以通过更改对话框的 BannerBitmap 属性来更改对话框中右上角所显示的图片。

18.3.14 安装过程中的自定义默认对话框

在定义安装过程的对话框时，除了可以使用上述 6 个默认的对话框外，Visual Studio 还提供了其他的自定义用户界面对话框，这些对话框可以在安装过程中显示以提供或收集信息。


在“用户界面编辑器”中，选择“安装”|“启动”节点，在快捷菜单上单击“添加对话框”命令，弹出如图 18.43 所示的“添加对话框”对话框。在此对话框中选择希望添加的对话框，单击“确定”按钮关闭对话框。使用此窗体主要可创建以下几种类型的对话框。



图 18.43 “添加对话框”对话框

- ☐ “单选按钮”用户界面对话框；
- ☐ “复选框”用户界面对话框；
- ☐ “客户信息”用户界面对话框；
- ☐ “文本框”用户界面对话框；
- ☐ “许可协议”用户界面对话框；
- ☐ “自述文件”用户界面对话框；
- ☐ “注册用户”用户界面对话框。

上面所提到的自定义对话框通常添加在程序安装的启动阶段，用于向用户提供应用程序信息或由用户进行安装信息设置。

 **注意：**“安装文件夹”对话框通常是“启动”部分的最后一个对话框，所以新添加的对话框要添加在这个对话框的前面。

下面将就这几种对话框进行简要的介绍。

18.3.15 “单选按钮”用户界面对话框

“单选按钮”对话框是一个提供了3种形式的对话框，分别可以提供2个、3个和4个互斥的单选项。“单选按钮”对话框的外观如图18.44所示。



图 18.44 “单选按钮”对话框

可以通过在对话框的属性窗中进行设置来改变对话框中各项的文本，下面以对话框中包含两个互斥项的对话框为例来说明如何对“单选按钮”用户界面对话框的属性进行设置。


BannerText 属性用于指示在对话框中标题栏的文本信息，在这个例子中对应的是“这里是标题栏文本”。在标题栏中所显示的文本的大小、字体和颜色都是固定的，程序可根据开发人员的输入自动确定文本的位置和换行情况。

BodyText 属性用于指示对话框主体部分的文本，在这个例子中，对应的是“这里是正文文本”。在对话框的主体部分所显示的文本的大小、字体和颜色都是固定的，程序可根据开发人员的输入自动确定文本的位置和换行情况。

Button1Label 属性用于指示对话框中第一个单选按钮旁的文本内容，在这个例子中，对应的是“第一选项”。在对话框的单选按钮旁所显示的文本的大小、字体和颜色都是固

定的，且开发人员所输入的字符串长度不能多于一行

Button1Value 属性是一个整数类型的属性值，表示第一个选项按钮的值。

说明：默认情况下，Button1Value 的属性值为 1，Button2Value 的属性值为 2，依此类推。

当用户做出了选择后，程序可以通过读取对话框的 ButtonProperty 值来确定用户所选的单选项。ButtonProperty 属性值为用户所选定的选项按钮的值，而 DefaultValue 值则用来指示程序默认选定的单选项，属性值是默认选定的选项按钮的值。

18.3.16 “复选框”用户界面对话框

“复选框”对话框一共提供了 3 种形式，在对话框中分别可以提供 2 个、3 个和 4 个复选项。“复选框”对话框的外观如图 18.45 所示。

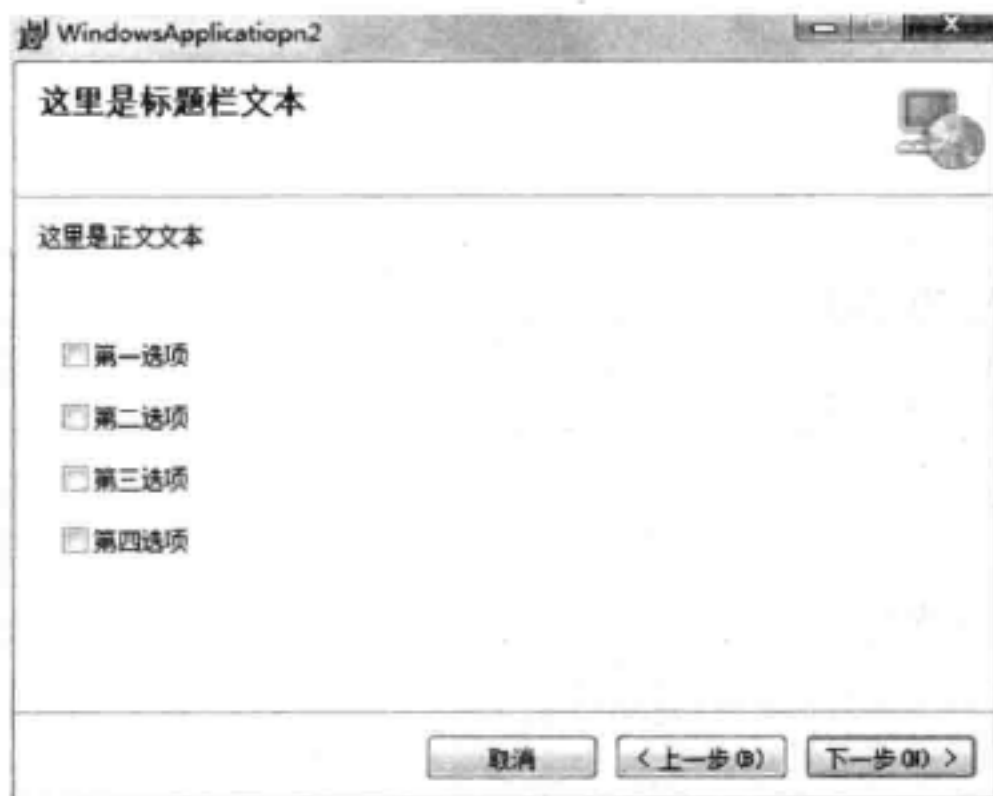


图 18.45 “复选框”对话框

可以通过在对话框的属性窗口中进行设置来改变对话框中各项的文本，下面以对话框中包含 4 个复选框的对话框为例来说明如何对“复选框”用户界面对话框的属性进行设置。

BannerText 属性用于指示在对话框中标题栏的文本信息，在这个例子中对应的是“这里是标题栏文本”。在标题栏中所显示的文本的大小、字体和颜色都是固定的，程序可根据开发人员的输入自动确定文本的位置和换行情况。


BodyText 属性用于指示对话框主体部分的文本，在这个例子中，对应的是“这里是正文文本”。同样，在对话框的主体部分所显示的文本的大小、字体和颜色都是固定的，程序可根据开发人员的输入来自动确定文本的位置和换行情况。

Checkbox1Label 属性用于指示对话框中第一个复选框旁的文本内容，在这个例子中，对应的是“第一选择”。同样，在对话框的单选按钮旁所显示的文本的大小、字体和颜色都是固定的，且开发人员所输入的字符串长度不能多于一行。

Checkbox1Value 属性的属性值只可能是 Checked 或 Unchecked，它用来表示第一个复选框的值，当第一个复选框的状态为被选中时，Checkbox1Value 的属性值为 Checked；当第一个复选框的状态为未被选中时，Checkbox1Value 的属性值为 Unchecked。

Checkbox1Visible 属性的属性值是一个 bool 类型的变量，用于指示第一个复选框是否

可见。通常情况下，此属性值设置为 true。

 说明：当用户做出了选择后，程序可以通过读取对话框中的每一个复选框所对应的 `Checkbox1Property` 值来确定此复选框的值在应用程序中所对应的属性值。

18.3.17 “文本框”用户界面对话框

“文本框”对话框是一个提供了 3 种形式的对话框，分别可以提供 2 个、3 个和 4 个文本框。“文本框”对话框的外观如图 18.46 所示。通过使用“文本框”用户界面对话框，可以在安装应用程序的过程中向用户显示文本输入字段，并且在安装过程中使用程序捕获这些字段的内容。



图 18.46 “文本框”对话框

可以通过在对话框的属性窗口中进行设置来改变对话框中各项的文本，下面以对话框中包含 4 个文本框的对话框为例来说明如何对“文本框”用户界面对话框的属性进行设置。


`BannerText` 属性用于指示在对话框中标题栏的文本信息，在这个例子中对应的是“这里是标题栏文本”。在标题栏中所显示的文本的大小、字体和颜色都是固定的，程序可根据开发人员的输入自动确定文本的位置和换行情况。

`BodyText` 属性用于指示对话框主体部分的文本，在这个例子中，对应的是“这里是正文文本”。同样地，在对话框的主体部分所显示的文本的大小、字体和颜色都是固定的，程序可根据开发人员的输入来自动确定文本的位置和换行情况。

`Edit1Label` 属性用于指示对话框中第一个文本框上的文本内容，在图 18.46 中，对应的是“第一个输入区域”。同样，在对话框的单选按钮旁所显示的文本的大小、字体和颜色都是固定的，且开发人员所输入的字符串长度不能多于一行。

`Edit1Value` 属性的属性值是一个字符串类型的变量，它用来表示第一个文本框的文本内容。

`Edit1Visible` 属性的属性值是一个 `bool` 类型的变量，用于指示第一个文本框是否可见，通常情况下，此属性值设置为 `true`。

 说明：当用户填写完毕后，程序可以通过读取对话框中的每一个文本框所对应的 Edit1Property 值，来确定此文本框中的文本在应用程序中所对应的属性值。

18.3.18 “客户信息”用户界面对话框

“客户信息”对话框的外观如图 18.47 所示。此对话框主要用于获取用户信息，其中包括：



图 18.47 “客户信息”对话框

- ☐ 用户名；
- ☐ 单位名称；
- ☐ 产品序列号。

开发人员可以通过设置 ShowOrganization 属性值来指示在此对话框中，是否需要用户填写单位名称信息，通过设置 ShowSerialNumber 属性值来指示在此对话框中，是否需要用户填写产品序列号信息。

在“客户信息”用户界面对话框中，用户所填写的信息将分别对应于安装程序的相应属性。具体对应关系如表 18.1 所示。

表 18.1 属性映射表

文 本 字 段	属 性 值	文 本 字 段	属 性 值
用户名	USERNAME	产品序列号	UserSID
单位名称	COMPANYNAME		

18.3.19 “许可协议”用户界面对话框

“许可协议”对话框的外观如图 18.48 所示，此对话框主要用于显示许可协议，并获得用户确认信息。在安装程序运行时，最终用户只有选择了“同意”单选框，安装程序才会继续执行。

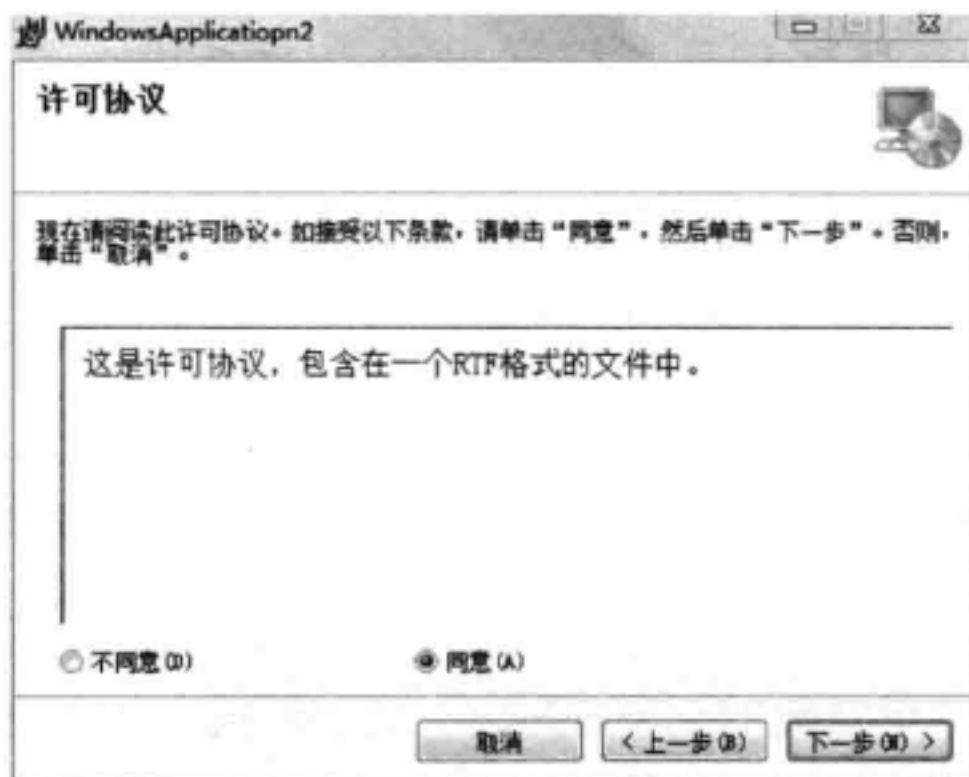



图 18.48 “许可协议”对话框

开发人员可以通过设置对话框的 LicenseFile 属性,指定一个加载许可协议文件的路径,安装程序在运行过程中会自动对此文件进行加载。

说明：“许可协议”文件必须为 RTF 格式。


18.3.20 “自述文件”用户界面对话框

“自述文件”对话框的外观如图 18.49 所示,此对话框主要用于显示程序的自述文件。



图 18.49 “自述文件”对话框

开发人员可以通过设置对话框的 ReadmeFile 属性,指定一个加载自述文件的路径,安装程序在运行过程中会自动对此文件进行加载。

说明：“自述文件”必须为 RTF 格式。

18.3.21 “注册用户”用户界面对话框

“注册用户”对话框的外观如图 18.50 所示,通过此对话框,用户可以通过单击“立即

注册”按钮，来启动由 Executable 属性所指定的可执行文件，从而提交注册信息。

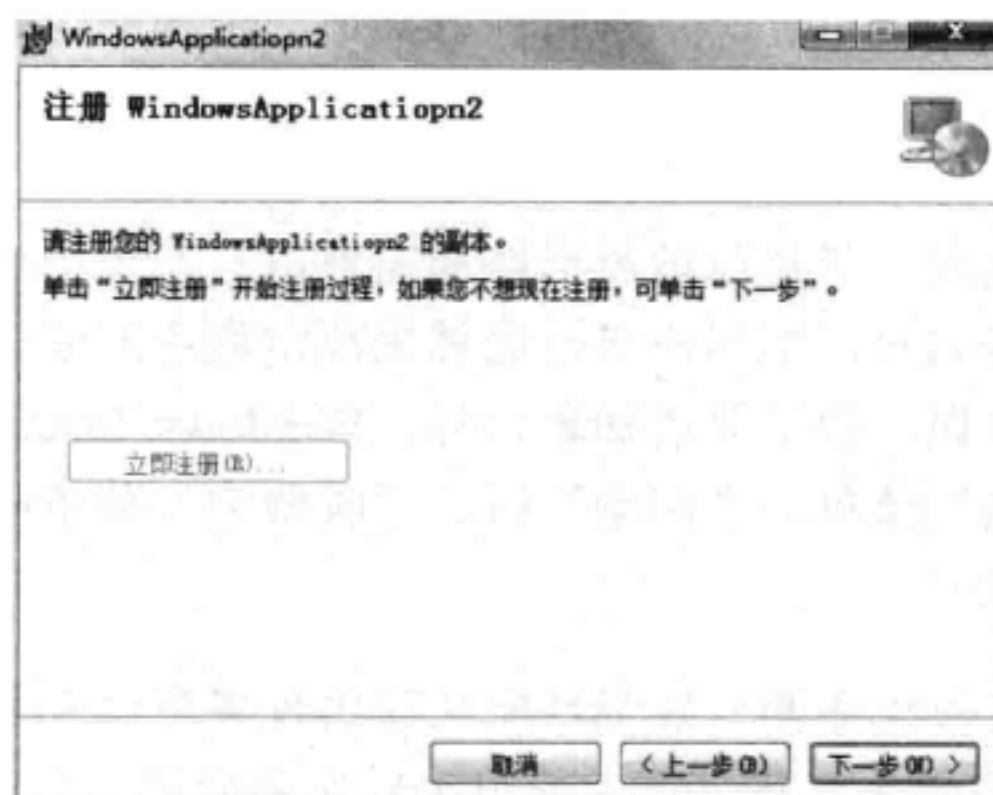


图 18.50 “注册用户”对话框

18.3.22 生成应用程序安装文件

在“解决方案资源管理器”中选择“WindowsApplication2 Installer”项目，在快捷菜单上选择“生成”命令，此时，将在默认文件夹（项目的 Debug 文件夹）中生成 WindowsApplication2 Installer.msi。将 WindowsApplication2 Installer.msi、Setup.exe，以及该目录下的其他所有文件和子目录复制到希望安装此应用程序的目标计算机上，通过双击 Setup.exe 运行安装程序。

 **注意：**必须在目标计算机上拥有安装权限才能运行该安装程序。

18.4 比较两种部署方案

在前面的各小节中，对于 Visual Studio 2010 中提供的两种部署方案进行了具体的说明，并针对具体案例进行了演示。但是在 Windows 应用程序的现实开发中，是使用 ClickOnce 部署方案还是使用 Windows Installer 部署方案对 Windows 应用程序进行部署，这个问题将在本节进行讨论。


在选择 Windows 应用程序的部署方案时，主要要考虑以下几点因素：

- ☐ 应用程序类型；
- ☐ 用户的类型和位置；
- ☐ 应用程序更新的频率；
- ☐ 安装要求。

当使用 ClickOnce 方案部署应用程序时，开发人员只需在需要创建安装程序的项目上进行发布设置，Visual Studio 2010 就会将此应用程序发布到指定的位置，而用户则只需在应用程序发布的指定位置中单击此安装程序的安装图标，则此应用程序将自动完成安装并运行。通过以上的步骤描述可以看出，ClickOnce 部署方案可以使开发人员和用户都轻松

地完成应用程序的发布和安装。除了运行和使用简单易行的优点，ClickOnce 应用程序还具有自动更新，安装后回滚，不影响共享组件或其他应用程序运行等优点。

当使用 Windows Installer 方案部署应用程序时，开发人员可以灵活地指定安装步骤。在各步中通过用户指定的选项，来决定应用程序的安装行为，使应用程序的安装具有更大的灵活性和用户的自定义性。但是这种灵活性却需要以开发人员更多的工作为代价，并且当用户没有经过良好的培训时，由用户自己选择来决定程序的安装属性，很有可能造成安装错误或安装不完全的失误。除了灵活部署以外，Windows Installer 应用程序还可以为多个用户使用，并向“开始”菜单、“启动”组、“收藏夹”菜单添加应用程序，并在应用程序安装时添加注册表访问。

 **注意：**.NET 提供的这两种部署方案都只能处理比较典型的部署需求，对于一些有特殊要求的应用程序部署，建议读者使用比较完善的第三方部署工具进行部署。

18.5 本章总结

在本章中，对于 Visual Studio 2010 中提供的两种部署方案进行了具体的说明和介绍，分别是 ClickOnce 部署方案和 Windows Installer 部署方案。并针对具体的案例对两种部署方案进行了演示。在本章的最后比较了两种部署方案的优缺点，并指出了每种部署方案的使用环境。最后需要指出的是，Visual Studio 2010 提供的这两种部署方案都只能处理比较典型的部署需求，对于一些有特殊要求的应用程序部署，建议读者使用比较完善的第三方部署工具进行部署。

18.6 实战练习

1. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序项目（或打开本书前面各章中创建的一个项目），然后使用 ClickOnce 部署方案发布本应用程序。
2. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序项目（或打开本书前面各章中创建的一个项目），然后在同一个解决方案中创建一个安装和部署项目，实现 Windows Installer 部署方案，并发布本应用程序。
3. 接第 2 题，在 Windows Installer 部署方案中增加自述文件、许可协议和客户信息等 3 个自定义对话框，设置好 3 个对话框中的相关属性，然后重新发布应用程序。

第 5 篇 C#高级编程技术 和工具

- ▶▶ 第 19 章 异常处理
- ▶▶ 第 20 章 文件系统与流
- ▶▶ 第 21 章 可扩展标记语言
- ▶▶ 第 22 章 多线程编程

第 19 章 异常处理

在本章中，重点对 C# 的异常处理机制进行介绍。将首先说明异常在程序设计过程中使用的意义，并且会对几种捕获异常的方法进行说明。接下来介绍 C# 编译器在异常处理过程中的一些特性和限制。最后对如何定义用户自己的异常进行介绍。

19.1 程序运行中的

简单地说，异常处理是一种处理 .NET 程序在运行过程中产生错误的机制。如果没有异常处理机制，可能会导致应用程序因发生错误而提前退出。如下述代码所示。

```
//声明使用的命名空间
using System;
namespace CSharpException1
{
    //定义 Test 类
    class Test
    {
        static void Main()
        {
            //调用 Divide() 方法
            double p = Divide(3, 5);
            Console.WriteLine(p);
        }
        //定义 Divide() 方法
        static double Divide(double a, double b)
        {
            double c = a/b;
            return c;
        }
    }
}
```


Test 类的 Divide() 方法将返回传入的两个参数相除之后的结果，并将输出如下结果：

```
0.6
```

但是，如果在 Main() 方法中像下面这样调用 Divide() 方法：

```
double p = Divide(3, 0);
```

在程序编译的时候，由于 Divide() 方法的两个参数都可以作为 double 型使用，因此不会产生编译错误。但是，在程序运行过程中，应用程序将试图用 3 除以 0，这个运算是无法进行的。因此，应用程序就会运行失败。

 **注意：** 解决这个问题的方法就是对 Divide() 方法传入的第 2 个参数进行合法性检查。如果该参数为 0，则返回参数错误的标志位。

```
static double Divide(double a, double b)
{
    //对分母 b 的值进行判断
    if(b == 0)
    {
        return 0;
    }
    double c = a/b;
    return c;
}
```


调用 Divide() 方法的函数可以进行如下处理：

```
static void Main()
{
    double p = Divide(3, 0);
    //对运算结果进行判断
    if(p == 0)
    {
        Console.WriteLine(" 输入错误 ");
    }
    else
    {
        Console.WriteLine(p);
    }
}
```

修改后的 Divide() 方法在接收到违法参数之后，将返回 0 以表示方法调用错误。但是，这样就存在一个问题，0 也可能是其他数值计算后的结果（如 0 除以 4）。在这种情况下，获得 Divide() 方法返回值后，就没有办法确定该值是正确的计算结果，还是方法调用错误标志。

```
static void Main()
{
    double p = Divide(0, 3);
    //对运算结果进行判断
    if(p == 0)
    {
        Console.WriteLine(" 输入错误!");
    }
    else
    {
        Console.WriteLine(p);
    }
}
```

上述程序运行之后，将返回“输入错误!”的错误提示，然而很明显，程序的运行是没有发生错误的。对于这个问题，即使采用其他错误标志值，这种情况还是有可能发生。那么如何才能够解决这个问题呢？可以通过使用 .NET 中的异常处理机制进行解决。

 **技巧：** 异常处理在程序调试阶段能给予很大的帮助，读者可以在以后的开发过程中仔细体会。

19.2 C#和.NET 中的异常处理

为了解决 19.1 节中所提出的问题，.NET 提出了一种解决方案——异常。程序设计人员可以定义一个异常，在出现不期望发生的事件的情况下抛出该异常。下面是 C#和.NET 中关于异常和异常处理的几点重要说明：

- 在.NET 中，所有的异常都是对象。.NET 中的异常共有一个基类，就是 `System.Exception` 类。在程序执行过程中，所有的方法都可以通过使用 C#中的关键字 `throw`，在发生不期望事件的时候抛出相应的异常。在外部调用方法中，可以使用 `try...catch` 结构获取和处理抛出的异常。
- 那些可能会抛出需要处理的异常的代码将放在 `try` 语段中，被称为试图捕获异常。而那些对所出现异常进行处理的代码会被放在紧邻 `try` 语段之后的 `catch` 语段中，被称为捕获异常。可以使用关键字 `catch` 加异常名，用来指定在 `catch` 语段中将处理的特定的异常类。此外，对一个 `try` 语段可以同时有多个 `catch` 语段用来捕获其抛出的异常，每个 `catch` 语段处理一种特定的异常类。
- 在 `try` 语段或者 `try...catch` 结构后还可以跟随 `finally` 语段，该语段的作用是用来放置那些必须被执行的代码，而不会受到是否产生异常的影响。
- 在程序执行的过程中，如果有异常发生，那么 `try` 语段内发生异常的语句后的代码将不会被执行，程序将执行有效的 `catch` 语段或者 `finally` 语段内的代码。
- 因为异常在.NET 中是以类和对象的形式实现的，所以它也遵循继承的原则。也就是说，如果有一个 `catch` 语段是用来处理基类异常，那么它就自动地可以处理所有该基类的子类异常。如果在捕获父类异常之后，又尝试捕获该父类异常的子类异常，那么将会引起编译器报错。
- `finally` 语段是可选的，异常的处理可以通过使用 `try...catch`、`try...catch...finally` 或者 `try...finally` 3 种结构中的任意一种。
- 如果有异常没有被捕获，那么 CLR 将捕获该异常，并终止程序的运行。

19.2.1 使用 `try...catch...finally` 结构处理异常

下面的例子演示 `try...catch` 结构的使用，具体代码如下所示。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _19._2._1
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            string s = null;
```



```

//使用 try...catch 结构处理异常
try
{
    //打印提示信息, 并进行字符串处理
    Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法前");
    Console.WriteLine("s.ToLower()");
    s.ToLower();
    Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法后");
}
//使用 catch 捕获异常
catch (NullReferenceException e)
{
    Console.WriteLine("进入 catch 语段...");
    Console.WriteLine("NullReferenceException 异常被捕获!");
}
Console.WriteLine("try...catch 语段后!");
}
}

```

在 Main()方法中设置字符串 s 的值为 null,之后调用了字符串对象 s 的 ToLower()方法,对 null 值进行处理。在这种情况下,CLR 会产生一个 NullReferenceException 异常。因为 ToLower()方法是在 try 语段中被调用的,所以会首先查找是否有 catch 语段负责捕获该异常。如果找到能够处理异常的 catch 语段,就会跳转到该 catch 语段继续运行。catch 语段的语法十分容易理解,跟在关键字 catch 后的参数是将要处理的目标异常的类型,在上面的示例中异常的类型是 NullReferenceException。代码执行后的输出结果如下:

```

在 try 语段中... 调用 s.ToLower() 方法前
进入 catch 语段...
NullReferenceException 异常被捕获!
try...catch 语段后!
按任意键继续...

```

对程序运行的输出结果和程序的源代码进行对比分析,可以发现在调用 s.ToLower()方法后产生了 NullReferenceException (空引用)异常,导致 try 语段中在其之后的代码没有得到执行,而是直接跳到 catch 语段继续运行。NullReferenceException 的产生,是由于程序尝试对 null 值进行非法操作。

对代码中的字符串 s 进行修改,使其具有一个明确的内容,代码如下:

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _19._2._1
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义字符串变量
            string s = "Hello World! ";
            //使用 try...catch 结构处理异常

```



```

try
{
    Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法前");
    Console.WriteLine("s.ToLower()");
    s.ToLower();
    Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法后");
}
//使用 catch 捕获异常
catch (NullReferenceException e)
{
    Console.WriteLine("进入 catch 语段...");
    Console.WriteLine("NullReferenceException Caught!");
}
Console.WriteLine("try...catch 语段后!");
}
}


```

修改之后的程序在运行过程中将不会产生任何异常，程序执行后将输出如下结果：

```

在 try 语段中... 调用 s.ToLower() 方法前
hello world!
在 try 语段中... 调用 s.ToLower() 方法后
try...catch 语段后!
按任意键继续...

```

 **技巧：**对于常见的异常读者应该尽量多了解。

19.2.2 捕获程序中可能产生的异常

上述修改后的代码，由于没有产生 `NullReferenceException` 异常，所以没有执行 `catch` 语段中的代码。但是，如果对 `try...block` 结构进行修改，并且仍设定字符串 `s` 的值为 `null`，将会产生何种结果呢？代码如下：

```

static void Main()
{
    Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法前");
    Console.WriteLine("s.ToLower()");
    s.ToLower()
    Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法后");
}


```

对上述代码进行编译和运行之后，将会输出如下结果：

```

调用 s.ToLower() 方法前
Unhandled Exception: System.NullReferenceException: Object reference not
set to an instance of an object.
at Test.Main() in c:\documents and settings\administrator\my
documents\visual studio projects \myconsole \class1.cs:line 8
按任意键继续...


```

 **注意：**由于 Visual Studio 的版本不同，所显示的异常信息也有细微差别。

之所以会产生如上结果，是因为没有对 `NullReferenceException` 异常进行处理。CLR

会终止程序的运行，并且发送如下信息报告。

- 异常消息：是对异常的描述。
- 程序执行异常的堆栈记录：对异常产生的跟踪。

 **说明：**在以上的例子中，异常的堆栈记录指出 Test 类中的 Main()方法在执行过程中产生异常，并且指明发生异常代码所在的文件和其完整路径，以及发生异常的代码所在的行数。

每个程序设计者都不希望应用程序由于产生异常而无法运行，所以会尝试捕获代码中所有可能发生的异常，并进行处理。仍以之前的代码为例，在产生 NullReferenceException 异常之后，可以使用 catch 语段进行捕获和处理。而且，还可以利用异常类的消息和堆栈记录属性，输出异常的具体信息和堆栈记录。实现代码如下所示。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _19._2._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义空字符串
            string s = null;
            //使用 try...catch 结构处理异常
            try
            {
                Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法前");
                Console.WriteLine("s.ToLower");
                s.ToLower();
                Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法后");
            }
            //使用 catch 捕获异常
            catch (NullReferenceException e)
            {
                //打印异常信息
                Console.WriteLine("\n 进入 catch 语段...");
                Console.WriteLine("NullReferenceException 异常被捕获!");
                Console.WriteLine("\n 异常信息");
                Console.WriteLine("=====");
                Console.WriteLine("e.Message");
                Console.WriteLine("\n 异常堆栈");
                Console.WriteLine("=====");
                Console.WriteLine("e.StackTrace");
            }
            Console.WriteLine("\ntry...catch 语段后!");
        }
    }
}
```


上述代码与之前示例代码的不同点在于，通过使用异常类的特性将异常的信息和堆栈记录进行打印。程序的执行结果如下：

```
在 try 语段中... 调用 s.ToLower() 方法前
进入 catch 语段...
NullReferenceException 异常被捕获!


异常信息
=====
Object reference not set to an instance of an object.

异常堆栈
=====
at Test.Main() in c:\documents and settings\administrator\my documents\
visual studio projects\myconsole\class1.cs:line 11

try...catch 语段后!
按任意键继续...
```

19.2.3 用 finally 语段释放资源

在 try 语段或者 catch 语段之后，是否使用 finally 语段是可选的。无论 try 语段在执行过程中是否产生异常，finally 语段内的代码都会运行。在 try 语段代码执行的过程中，存在由于发生异常而导致申请资源无法得到释放的可能。这种情况下，就可以使用 finally 语段释放申请的资源。

 **技巧：**finally 语段通常用来关闭文件、数据库、socket 连接或者其他在 try 语段中打开的资源。

以如下程序作为示例：

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _19._2._3
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义字符串变量
            string s = "Hello World!";
            //使用 try...catch 结构处理异常
            try
            {
                Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法前");
                Console.WriteLine("s.ToLower");
                s.ToLower();
            }
        }
    }
}
```



```

        Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法后");
    }
    //使用 catch 捕获异常
    catch (NullReferenceException e)
    {
        Console.WriteLine("\n 进入 catch 语段...");
        Console.WriteLine("NullReferenceException 异常被捕获!");
    }
    //使用 finally 进行必要的处理过程
    finally
    {
        Console.WriteLine("\n 进入 finally 语段...");
    }
}
}
}

```

程序执行后的输出结果如下:

```

在 try 语段中... 调用 s.ToLower() 方法前
hello world!
在 try 语段中... 调用 s.ToLower() 方法后
进入 finally 语段...
按任意键继续...

```

因为程序执行过程中没有异常发生, `finally` 语段的代码将在 `try` 语段执行完成后被执行。如果将 `Main()` 方法中的字符串 `s` 的值设成 `null`, 那么执行时将产生异常, 代码如下:

```

static void Main()
{
    string s = null;
    ...
}

```

输出结果将变成:

```

在 try 语段中... 调用 s.ToLower() 方法前

进入 catch 语段...
NullReferenceException 异常被捕获!

进入 finally 语段...
按任意键继续...

```

通过对上述两种输出结果的对比可以看出, 无论 `try...catch` 结构在执行过程中是否会产生异常, `finally` 语段中的代码都将得到执行。在程序设计中, 也可以直接将 `finally` 语段与 `try` 语段连用, 而不使用 `catch` 语段。具体如以下代码所示。

```

static void Main()
{
    string s = "Hello World!";
    //单独使用 try 语段
    try
    {
        Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法前");
        Console.WriteLine("s.ToLower");
    }
}

```

```

        Console.WriteLine("在 try 语段中... 调用 s.ToLower() 方法后");
    }
    //使用 finally 进行处理
    finally
    {
        Console.WriteLine("\n 进入 finally 语段...");
    }
}

```


程序运行后，将输出结果：

```

在 try 语段中... 调用 s.ToLower() 方法前
hello world!
在 try 语段中... 调用 s.ToLower() 方法后

进入 finally 语段...
按任意键继续...

```

 说明：finally 语段的代码在 try 语段执行完之后，即使没有 catch 语段也会得到运行。

19.3 多异常的捕获

19.3.1 什么多异常

有的时候，try 语段的代码在执行时可能会抛出多个不同的异常，这就需要有一系列的 catch 语段用来捕获这些异常。例如，在一段程序中可以有 3 个不同的 catch 语段，第 1 个用来捕获 `NullReferenceException` 异常；第 2 个捕获 `IndexOutOfRangeException` 异常；而第 3 个用来捕获其他异常。其中，`IndexOutOfRangeException` 异常是由于所访问的数组元素不在数组界限内产生的。

多异常处理如下代码所示。

```

//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _19._3._1
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义字符串变量
            string s = "Hello Word!";
            int []i = new int[3];
            //使用 try...catch 结构处理异常
            try
            {
                Console.WriteLine("进入 try 语段... \n");
                //可能引发 NullReferenceException 异常
            }
        }
    }
}

```



```


        Console.WriteLine("将字符串小写:" + s.ToLower());
        //可能引发 NullReferenceException 异常和
        //IndexOutOfRangeException 异常
        Console.WriteLine("数组中的第一个元素是" + i[0].ToString());
        //可能引发除数为 0 的异常
        i[0] = 3;
        i[1] = 4/i[0];
        Console.WriteLine("\n 离开 try 语段...");
    }
    //使用 catch 捕获不同的异常
    catch (NullReferenceException e)
    {
        Console.WriteLine("\n 进入 catch 语段...");
        Console.WriteLine("NullReferenceException 异常被捕获!");
    }
    catch (IndexOutOfRangeException e)
    {
        Console.WriteLine("\n 进入 catch 语段...");
        Console.WriteLine("IndexOutOfRangeException 异常被捕获!");
    }
    catch (Exception e)
    {
        //打印异常信息
        Console.WriteLine("\n 进入 catch 语段...");
        Console.WriteLine("其他异常被捕获!");
        Console.WriteLine(e.Message);
    }
}
}
}

```

在上述代码中, 定义了字符串 `s` 和整数数组 `i`, 其中数组 `i` 中有 3 个元素。代码在执行过程中, 有 3 个地方可能会产生异常。

- ❑ 如果 `s` 或者 `i` 为空, 则会产生 `NullReferenceException` 异常;
- ❑ 在访问数组 `i` 的时候, 可能会产生 `IndexOutOfRangeException` 异常;
- ❑ 在 4 除以 `i[0]` 的过程中, 如果 `i[0]` 为 0, 则会产生 `DivideByZeroException` 异常。

在程序中, 声明了 3 个 `catch` 语段用来捕获这 3 种不同的异常。其中, 最后一个 `catch` 语段设计成可以捕获除了 `NullReferenceException` 和 `IndexOutOfRangeException` 两种异常外的其他异常, 前两种异常已经由前两个 `catch` 语段进行处理。

 **注意:** 上文所提到的 3 个异常只有一个会发生, 因为一个异常发生后 `try` 语段就会停止执行, 而跳转到 `catch` 语段执行。

上段代码执行后, 会输出如下结果:

```

进入 try 语段...

将字符串小写: hello world!
数组中的第一个元素是: 0

离开 try 语段...
按任意键继续...

```

上述代码在执行过程中，不会产生任何异常。如果将字符串 `s` 的值设为 `null`，则程序运行后将输出什么结果呢？

```
static void Main()
{
    string s = null;
    ...
}
```

输出结果是：

进入 try 语段...

进入 catch 语段...

NullReferenceException 异常被捕获！

按任意键继续...

程序在调用 `ToLower()` 方法时产生异常，直接跳到 `catch` 语段执行，而不再执行 `try` 语段内的其他代码和其他的 `catch` 语段。现在，将字符串 `s` 设为原值，但是尝试输出数组 `i` 的第 5 个元素，代码如下：

```
Console.WriteLine(" 数组中的第五个元素是 " + i[5].ToString);
```

输出结果将变成：

进入 try 语段...

将字符串小写：hello world!

进入 catch 语段...

IndexOutOfRangeException 异常被捕获！

按任意键继续...

验证完以上两种情况后，再对第 3 种情况进行验证。将 `i[0]` 的值设为 0，使程序产生 `DivideByZeroException` 异常。

```
...
i[0] = 0;
...
```

程序运行后输出结果是：

进入 try 语段...

将字符串小写：hello world!

数组中的第一个元素是：0


DivideByZeroException

进入 catch 语段...

其他异常被捕获！


按任意键继续...

上述代码在执行 `3/i[0]` 的时候，将产生 `DivideByZeroException` 异常。

 **注意：**程序中是使用 Exception 异常类捕获该异常。Exception 异常是.NET 中所有异常的基类，如果产生没有被其他 catch 语段捕获的异常，都可以被负责捕获 Exception 异常的 catch 语段处理。

19.3.2 异常的继承关系

在 19.2 节中已经提到过，由于在.NET 中异常是以类和对象的方式实现的，所以异常也遵循类的继承原则。也就是说，如果一个 catch 语段是用来处理基类异常，那么它就将自动地可以处理所有该基类的子类异常。

 **说明：**在捕获父类异常之后，如果尝试捕获该父类异常的子类异常，则会引起编译器报错。

```
//声明使用的命名空间
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _19._3._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //定义空字符串
            string s = null;
            //使用 try...catch 结构处理异常
            try
            {
                Console.WriteLine("进入 try 语段...\n");
                //可能引发 NullReferenceException 异常
                Console.WriteLine("将字符串小写:" + s.ToLower());
                Console.WriteLine("\n 离开 try 语段...");
            }
            //使用 catch 捕获异常
            catch (Exception e)
            {
                //打印异常信息
                Console.WriteLine("\n 进入 catch 语段...");
                Console.WriteLine("其他异常被捕获!");
                Console.WriteLine(e.Message);
            }
            catch (NullReferenceException e)
            {
                Console.WriteLine("\n 进入 catch 语段...");
                Console.WriteLine("NullReferenceException 异常被捕获!");
            }
        }
    }
}
```


```

    }
}

```

上述代码中，Exception 异常的处理在其子异常 NullReferenceException 异常之前。这种情况下，无论 try 语段中产生什么异常，都会被第一个 catch 语段捕获，从而导致第二个 catch 语段内的代码永远都不会被执行，成为无用代码。在对程序编译的时候，编译器会对此进行检查，并报告错误。

上一个 catch 子句已经捕获了此类型或超类型(“System.Exception”)的所有异常。

 注意：在使用多个 catch 语段的时候，必须要考虑异常之间的继承关系。

19.3.3 捕获所有异常的方法

catch 语段的参数可以只写所处理异常的类名，而不用规定具体对象，例如：

```

catch (NullReferenceException)
{
    ...
}

```

但是，在实际使用中不建议使用这种方法。

在使用 catch 语段的时候，还可以只写出关键字 catch，其后不加具体参数。这样的话，就相当于捕获所有产生的异常，与捕获 Exception 异常类似。

```

catch
{
    ...
}

```

不过，特别不推荐使用这种方法，如果想要捕获所有异常，可以使用 Exception 异常类。

19.4 定义用户异常的方法

在 C# 中，允许用户自定义异常类型，用来表示用户代码在执行过程中出现不期望发生的事件。前面已经介绍过，C# 中异常是以类和对象的方式实现的。所以，在定义新的用户异常的时候，就需要为这个异常定义一个新类。在介绍用户如何自定义异常之前，需要先了解 .NET Framework 下异常的继承关系。

在 .NET 中，异常可以分为以下两个主要的类别。

- ❑ System Exception：由运行平台（Common Language Runtime）产生的异常；
- ❑ Application Exception：由用户程序产生的异常。

如图 19.1 所示，该图简要地说明了 .NET Framework 中异常的继承关系。

由该图可知，所有用户自定义的异常都来自 ApplicationException 类，下面以 InvalidArgumentException 异常的定义来说明用户如何自定义异常。前面的内容介绍过 Divide() 方

法，调用该方法的时候如果第二个参数是 0，则会发生运行错误而抛出 `DivideByZeroException` 异常。

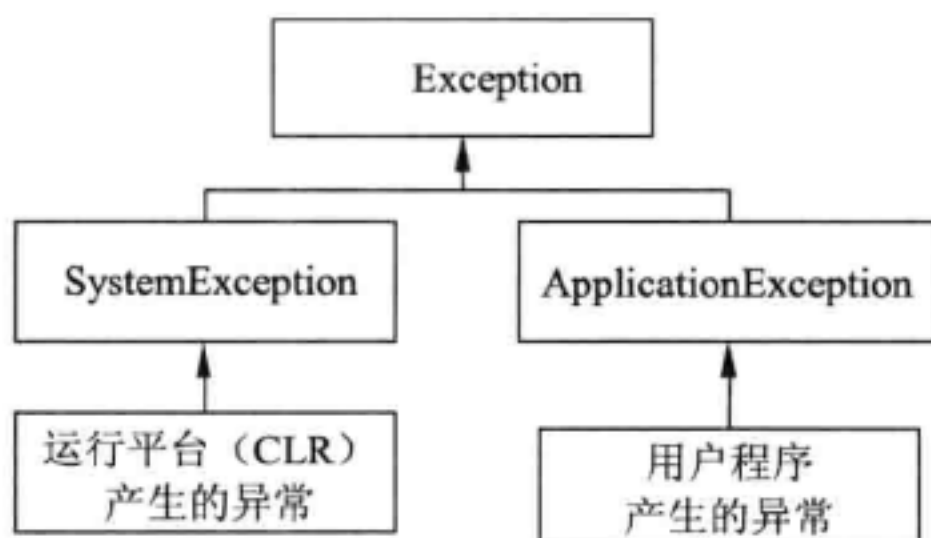



图 19.1 .NET Framework 中的异常继承关系

现在，设计 `Divide()` 方法在调用过程中发生错误时，将抛出用户自定义的 `InvalidArgumentException` 异常。

`InvalidArgumentException` 异常定义如下：

```
//定义继承 ApplicationException 类的 InvalidArgumentException 类
class InvalidArgumentException : ApplicationException
{
    public InvalidArgumentException() : base(" Divide By Zero Error ")
    {
    }
    public InvalidArgumentException(string Message) : base(message)
    {
    }
}
```


正如所有用户自定义异常一样，`InvalidArgumentException` 类也是在继承 `ApplicationException` 异常的基础上定义的。在 `InvalidArgumentException` 类中有两个构造函数，一个是无参数的构造函数，另一个是以字符串 `message` 为参数的构造函数。两个构造函数都是将相应的字符串信息传给基类的构造函数，从而用来初始化基类的相关信息。用户自定义完异常之后，就可以对其进行调用。

 **技巧：**用户定义的所有异常都要继承自 .NET Framework 中的异常类。

在设计方法函数的时候，可以使用关键字 `throw` 抛出异常。现在以 `Divide()` 方法为例，演示方法在设计过程中如何抛出异常，示例代码如下：

```
static double Divide(double a, double b)
{
    //对分母 b 的值进行判断，抛出异常
    if(b == 0)
    {
        throw new InvalidArgumentException();
    }
    double c = a/b;
    return c;
}
```

在此段代码中，由于 `Divide()` 方法的第二个参数为 0，所以在执行过程中会抛出 `InvalidArgumentException` 异常。

 注意：`Divide()` 方法抛出的异常是新创建的 `InvalidArgumentException` 类的对象。

在使用过程中，还可以给抛出的异常定义自己所需的异常信息，如下所示。


```
throw new ArgumentException("错误：除法的第2个参数为0");
```

下面将在 `Main()` 方法中使用 `Divide()` 方法，并用 `try...catch` 结构捕获发生的异常，示例代码如下：

```
static void Main(string[] args)
{
    //使用 try...catch 结构处理异常
    try
    {
        Console.WriteLine("进入 try 语段...");
        double d = Divide(3, 0);
        Console.WriteLine("\t 除法的结果: {0}", d);
    }
    //使用 catch 捕获异常
    catch (ArgumentException e)
    {
        //打印异常信息
        Console.WriteLine("\n 进入 catch 语段...");
        Console.WriteLine("\n System.InvalidArgumentException 异常被捕获...");
        Console.WriteLine("\n 错误: " + e.Message);
    }
}
```

上述代码与之前的例子比较相似，但是执行时抛出和捕获的异常是用户自定义的异常。程序在执行之后，将输出如下内容：

```
进入 try 语段...
进入 catch 语段...
System.InvalidArgumentException 异常被捕获...
错误: Divide By Zero Error
按任意键继续...
```

 说明：上述代码执行后，返回的异常消息是用户定义的信息，这对自定义异常的使用非常重要。

下面对用户自定义异常的要点进行说明：

- ❑ 在 .NET Framework 中，所有异常的名字都是以 `Exception` 结尾，例如，`SystemException`，`NullReferenceException` 和 `IndexOutOfRangeException` 等。因此，用户对自定义异常的使用最好也遵守这一惯例。
- ❑ 通常情况下，用户定义的异常都是通过继承 `ApplicationException` 类实现的。
- ❑ 由于抛出和捕获异常在执行时具有较大的优先权，即有异常产生就会立刻执行异常处理过程，所以在程序设计过程中最好不要使用过多的不需要的异常。

19.5 本章总结

在本章中的开始部分，主要介绍了一些异常的基本概念，以及如何处理异常的操作，并简单介绍了如何处理多异常的操作。最后，将异常作为类进行研究，描述了异常中信息的读取方法，以及开发人员如何自定义异常。

19.6 实战练习

1. 在 Visual Studio 2010 中输入以下代码，运行的结果是什么？

```
public class ExceptionApp {  
    public static void ThrowException(){throw new Exception();}  
    public static void Main()  
    {  
        try  
        {  
            ThrowException();  
            Console.WriteLine("try");  
        }  
        catch(Exception e) {  
            Console.WriteLine("catch");  
        }  
        finally  
        {  
            Console.WriteLine("finally");  
        }  
    }  
}
```

2. 在 Visual Studio 2010 中新建一个控制台应用程序，定义一个异常类 DivZero，当除法运算中除数为 0 时，用 DivZero 类显示“除数不能为 0”的异常信息。再定义一个测试类测试该自定义异常。

第 20 章 文件系统与流

在本章中，将讲解如何操作 Windows 的文件系统，研究如何操作物理驱动器、文件夹和文件并对它们执行不同的操作。还将介绍在 C#以及.NET 中涉及的不同类型的流，并且讲解如何从文件中读取数据和写入数据的操作。并将学习对象序列化的概念、如何使用 C#将对象序列化，以及异步输入/输出的相关操作。

20.1 软件系统环境相关信息

在本节中，将讲解如何获得软件系统环境的相关信息，例如，可获得 Windows 系统文件夹或程序文件夹的路径、当前用户名称、操作系统的版本等。

20.1.1 用 System.Environment 类获得应用程序运行环境的信息

System.Environment 类是用于包含应用程序运行环境的相关信息的基础类。在这个类中提供了用于获得 Windows 系统文件夹或程序文件夹的路径、当前用户名称、操作系统版本等信息的属性和方法，关于这个类的基本属性和方法的描述如表 20.1 所示。

表 20.1 System.Environment类的基本属性和方法

成 员	描 述
CurrentDirectory	程序开始的文件夹名称
MachineName	当前运行的计算机的名称
OSVersion	当前操作系统的版本信息
UserName	当前操纵程序的用户的用户名称
Version()	返回一个System.Version对象，描述CLR的完整的版本信息
Exit()	停止当前的程序，对操作系统返回终止信息
GetFolderPath()	获得Windows操作系统的各种标准文件夹的完整路径，例如程序文件、我的文档、开始菜单等
GetLogicalDrives()	返回一个字符串格式的序列，包含当前系统的各种驱动器的列表

20.1.2 System.Environment 类的应用举例

下面，将用 20.1.1 节中所提到的 System.Environment 类的属性和方法做一个示范程序来展示环境变量的相关信息。

在这个示例程序中，Windows 窗体包含一个名为 lbx 的 Listbox 控件来展示 Windows

系统环境的相关信息，一个名为 btnGo 的 Button 控件用来开始捕获信息，一个名为 btnExit 的 Button 控件用来结束应用程序。

下面，按以下步骤创建 Windows 应用程序。

(1) 创建一个 Windows 应用程序，在 Windows 窗体的相应属性窗口中，将此窗体的 name 属性设置为 formEnviromentInformation，text 属性设置为“软件系统的环境信息”。

(2) 向窗体中添加 ListBox 控件，将 ListBox 控件命名为 lbx，在属性窗口中，将 ListBox 控件的 Anchor 属性设置为“Top, Left, Right”。


(3) 向窗体中添加 Button 控件，将 Button 控件命名为“btnGo”，在属性窗口中，将此 Button 控件的 text 属性设置为“运行”，Anchor 属性设置为“Bottom, Right”。

(4) 向窗体中添加另一个 Button 控件，将 Button 控件命名为“btnExit”，在属性窗口中，将此 Button 控件的 text 属性设置为“退出”，Anchor 属性设置为“Bottom, Right”。

双击名为“btnGO”的 Button 控件，在程序中自动添加“运行”按键的 Click 事件。在此事件中添加相应的程序处理代码，如下所示。


btnGo_Click 事件处理函数：

```
//定义 btnGo_Click() 事件处理函数
private void btnGo_Click(object sender, EventArgs e)
{
    //定义变量
    OperatingSystem os = Environment.OSVersion;
    PlatformID OSid = os.Platform;
    //调用 Environment.GetLogicalDrives() 方法
    string[] drives = Environment.GetLogicalDrives();
    string drivesString = "";
    //使用 foreach 循环遍历 drives 内数据
    foreach (string drive in drives)
    {
        drivesString += drive + ",";
    }
    drivesString = drivesString.TrimEnd(' ', ',');
    //打印输出信息
    lbx.Items.Add("计算机的名称: \t" + Environment.MachineName);
    lbx.Items.Add("操作系统版本信息: \t" + Environment.OSVersion);
    lbx.Items.Add("操作系统 ID: \t" + OSid);
    lbx.Items.Add("当前文件夹: \t" + Environment.CurrentDirectory);
    lbx.Items.Add("CLR 的版本信息: \t" + Environment.Version);
    lbx.Items.Add("驱动器列表: \t" + drivesString);
}
```

 说明：在上面的事件处理程序中，Environment.GetLogicalDrives() 返回了一个字符串序列，每一个字符串展示了一个驱动器的名字，可通过 foreach 语句遍历字符串序列从而获得每一个驱动器的名字。Environment.Version 返回一个 System.Version 对象，包含了公共语言运行时的当前版本的细节信息。

btnExit_Click 事件处理函数的代码设置如下：

```
private void btnExit_Click(object sender, EventArgs e)
{
    Environment.Exit(0);
}
```

 **注意：**在上面的事件处理程序中，调用了 `Environment` 类的 `Exit()` 方法来结束当前程序。

在 `btnExit_Click` 事件处理函数中，程序传递了参数 0 给 `Exit()` 方法。这个值将会返回给操作系统，并且用来检查程序是否成功地结束。

上段示例程序通过公共属性和方法重新得到了简单的环境变量信息，并把它们添加到了 `Listbox` 里。当对系统执行这个程序时，可得到如图 20.1 所示的结果。



图 20.1 程序环境信息获取界面

20.1.3 用 `Environment.GetFolderPath()` 获得各种 Windows 标准文件夹的路径

`Environment.GetFolderPath()` 方法可以用来获得当前机器的各种 Windows 标准文件夹的完整的路径。传递给该方法的唯一参数是来自于 `System.Environment.SpecialFolder` 的枚举值的变量。该枚举值较常用的成员如表 20.2 所示。

表 20.2 `System.Environment.SpecialFolder` 的枚举值

成 员	描 述
<code>ProgramFiles</code>	通常程序所安装在的program files文件夹
<code>CommonProgramFiles</code>	程序文件的通用文件夹
<code>DesktopDirectory</code>	展示用户桌面的文件夹
<code>Favorites</code>	Favorites文件夹用来储存最喜欢的链接
<code>History</code>	History文件夹来储存历史文件
<code>Personal</code>	我的文档文件夹
<code>Programs</code>	用于展示开始菜单里程序的文件夹
<code>Recent</code>	最近文档的文件夹
<code>SendTo</code>	发送到文件夹
<code>StartMenu</code>	开始菜单文件夹
<code>Startup</code>	展示开始菜单的所有程序的文件夹
<code>System</code>	Windows文件夹的System文件夹
<code>ApplicationData</code>	应用程序的数据文件夹
<code>CommonApplicationData</code>	通用应用程序的数据文件夹
<code>LocalApplicationData</code>	本地程序数据文件夹
<code>Cookies</code>	用于储存cookies设置的文件夹

可以使用一个简单的程序来理解它们的功能，通过更改 20.1.2 节程序中的 `btnGo_Click` 事件来增添这些结果，代码如下：


```

//定义 btnGo_Click() 事件处理函数
private void (object sender, EventArgs e)
{
    //定义变量
    OperatingSystem os = Environment.OSVersion;
    PlatformID OSid = os.Platform;
    //调用 Environment.GetLogicalDrives() 方法
    string[] drives = Environment.GetLogicalDrives();
    string drivesString = "";
    //使用 foreach 循环遍历 drives 内数据
    foreach (string drive in drives)
    {
        drivesString += drive + ",";
    }
    drivesString = drivesString.TrimEnd(' ', ',');
    //打印输出信息
    lbx.Items.Add("计算机的名称: \t" +
        Environment.MachineName);
    lbx.Items.Add("操作系统版本信息: \t" + Environment.OSVersion);
    lbx.Items.Add("操作系统 ID: \t" + OSid);
    lbx.Items.Add("当前文件夹: \t" + Environment.CurrentDirectory);
    lbx.Items.Add("CLR 的版本信息: \t" + Environment.Version);
    lbx.Items.Add("驱动器列表: \t" + drivesString);
    lbx.Items.Add("Program Files: \t" +
        Environment.SpecialFolder.ProgramFiles);
    lbx.Items.Add("Common Program Files: \t" +
        Environment.SpecialFolder.CommonProgramFiles);
    lbx.Items.Add("Desktop Directory: \t" +
        Environment.SpecialFolder.DesktopDirectory);
    lbx.Items.Add("Favorites: \t" +
        Environment.SpecialFolder.Favorites);
    lbx.Items.Add("History: \t" +
        Environment.SpecialFolder.History);
    lbx.Items.Add("Personal: \t" +
        Environment.SpecialFolder.Personal);
    lbx.Items.Add("Programs: \t" +
        Environment.SpecialFolder.Programs);
    lbx.Items.Add("Recent: \t" +
        Environment.SpecialFolder.Recent);
    lbx.Items.Add("Send To: \t" +
        Environment.SpecialFolder.SendTo);
    lbx.Items.Add("Start Menu: \t" +
        Environment.SpecialFolder.StartMenu);
    lbx.Items.Add("Startup: \t" +
        Environment.SpecialFolder.Startup);
    lbx.Items.Add("System: \t" +
        Environment.SpecialFolder.System);
    lbx.Items.Add("Application Data: \t" +
        Environment.SpecialFolder.ApplicationData);
    lbx.Items.Add("Common Application Data: \t" +
        Environment.SpecialFolder.CommonApplicationData);
    lbx.Items.Add("Local Application Data: \t" +
        Environment.SpecialFolder.LocalApplicationData);
    lbx.Items.Add("Cookies: \t" +
        Environment.SpecialFolder.Cookies);
}

```

程序的运行结果如图 20.2 所示。



图 20.2 程序环境信息获取界面

在上面的示例中，将具体指出在操作系统中的每一个 `Environment.SpecialFolder` 枚举值所对应的文件夹。

20.2 对文件进行操作

本节将学习如何执行对指定文件的操作，例如文件的复制、移动、删除和重命名。对文件的相关操作，主要通过使用 `System.IO.File` 类和 `System.IO.FileInfo` 类来进行。

20.2.1 C#对文件进行操作的类

在.NET中可以通过使用 `System.IO.File` 类和 `System.IO.FileInfo` 类对操作系统中的文件进行维护及各种其他操作。其中，`System.IO.File` 类包含各种可以对文件进行操作的静态成员方法，而 `System.IO.FileInfo` 类则是需要通过实现特定的对象以对某个文件的属性进行操作。

20.2.2 用 `System.IO.File` 类的静态方法操作文件

如 20.2.1 节中所提到的，`System.IO.File` 类包含各种可以对文件进行操作的静态成员方法，当使用这些方法时，不需要对类进行实例化便可直接引用。`File` 类中的常用静态方法如表 20.3 所示。

表 20.3 `System.IO.File`类中的静态方法

成员方法	描述
<code>Copy()</code>	将特定的文件复制到指定的路径下
<code>Create()</code>	创建指定文件
<code>Delete()</code>	删除指定文件
<code>Exists()</code>	检验指定的文件是否存在，返回值为boolean类型

续表


成 员 方 法	描 述
GetAttributes()	返回包含文件相关属性信息的System.IO.FileAttributes类型对象, 例如文件是否是隐藏文件等
GetCreationTime()	返回包含文件创建时间的DateTime类型对象
GetLastAccessTime()	返回包含最后一次访问文件时间的DateTime类型对象
GetLastWriteTime()	返回包含最后一次修改文件时间的DateTime类型对象
Move()	将文件移到指定的路径下
Open()	打开指定的文件, 并返回该文件的System.IO.FileStream对象
OpenRead()	以只读的方式打开指定文件, 返回该文件只读的System.IO.FileStream对象
OpenWrite()	打开指定的文件, 并可以对文件进行读写操作, 返回该文件可以进行读写的System.IO.FileStream对象
SetAttributes()	通过传入包含文件属性信息的System.IO.FileAttributes类型对象, 设定文件的属性

上述这些对文件的操作方法都可以直接使用, 但是很难在一个示例程序中将这些方法的使用和输出结果都表示出来。所以在接下来的内容中, 将对各个方法在程序中的用法分别说明。

□ 使用 Create()方法创建文件

如果想要在 C 盘的根目录下建立一个名为 c-sharp.txt 的文件, 可以进行如下定义:

```
File.Create("C:\\c-sharp.txt");
```

 **注意:** 如果在程序设计过程中使用了对文件方法的调用, 那么就需要在代码的开始处添加对 System.IO 命名空间的引用。


□ 使用 Copy()/Move()方法复制/移动文件

如果想要把上面创建的 c-sharp.txt 文件复制到 C:\\my programs 目录中, 可以进行如下操作:

```
File.Copy("C:\\c-sharp.txt", "C:\\my programs\\c-sharp.txt");
```

类似的, 还可以使用 Move()方法实现对文件的移动。另外, 还可以使用 Copy()和 Create()的重载方法。使用 Copy()的重载方法如下所示。

```
File.Copy("C:\\c-sharp.txt", "C:\\my programs\\c-sharp.txt", true);
```

 **技巧:** 如果在目标路径下, 已经存在与要复制或者创建文件同名的文件, 将返回一个 boolean 类型的变量进行提示。

□ 使用 Exists()方法检查文件

如果想了解目录下是否存在指定文件, 可以使用 Exists()方法。使用方法如下:

```
if(!File.Exists("C:\\c-sharp.txt"))
{
    File.Create("C:\\c-sharp.txt");
}
```

□ 使用 GetAttributes()方法获得文件属性

在 File 类中, 可以使用 GetAttributes()方法获得指定文件的属性信息, 使用方法如下:

```
FileAttributes attrs = File.GetAttributes("C:\\c-sharp.txt");
lbx.Items.Add("File 'C:\\c-sharp.txt'");
lbx.Items.Add(attrs.ToString());
```

执行上述代码之后，可以得到文件的属性信息，例如该文件是否为一份存档文件等。同样地，在使用中还可以利用 FileAttributes 的各种文件属性对文件进行设置。

20.2.3 用 System.IO.FileInfo 类的方法操作文件

同样地，System.IO.FileInfo 类也提供了多种对文件的操作。如表 20.4 所示，对 FileInfo 类中的重要方法和属性进行了描述。


 **注意：**与 File 类不同的是，在使用 FileInfo 类的方法之前，需要先创建一个 FileInfo 类的对象。

表 20.4 System.IO.FileInfo 类中的成员

成 员	描 述
CreationTime	文件的创建时间
Directory	返回包含文件所在目录（文件夹）信息的DirectoryInfo对象
DirectoryName	文件所在目录的名称
Extension	文件的扩展类型（如：.exe, .cs, .aspx）
FullName	文件的全称，即具体路径和名字（如：C:\c-sharp.txt）
LastAccessTime	包含文件最后一次访问时间信息的DateTime对象
LastWriteTime	包含文件最后一次修改时间信息的DateTime对象
Length	文件的大小（所占字节数）
Name	文件的名称（如：c-sharp.txt）
CopyTo()	将文件复制到指定的路径下
Creat()	创建文件
Delete()	删除文件
Exists()	返回boolean类型数值标识，指定文件是否存在
MoveTo()	将文件移动到指定的路径下
Open()	以可读写的方式打开该文件
OpenRead()	以只读的方式打开指定文件，返回该文件只读的System.IO.FileStream对象
OpenWrite()	打开指定的文件，并可以对文件进行读写操作，返回该文件可以进行读写的System.IO.FileStream对象
OpenText()	打开指定的文件，并返回以UTF8格式的System.IO.StreamReader对象

下面通过程序介绍上述 FileInfo 类属性和方法的具体使用方法。程序对文件进行不同的操作，并将执行的结果输出，实现代码如下：

```
private void filePrint()
{
    //获得文件信息
    FileInfo file = new FileInfo("C:\\c-sharp.txt");
    //打印输出信息
    System.Console.WriteLine ("文件名: " + file.Name);
}
```



```

System.Console.WriteLine ("文件扩展名: " + file.Extension);
System.Console.WriteLine ("文件全称: " + file.FullName);
System.Console.WriteLine ("所在目录: " + file.DirectoryName);
System.Console.WriteLine ("文件大小: " + file.Length.ToString() +
    "bytes");
System.Console.WriteLine ("文件属性: " + file.Attributes.ToString());
}

```

上述代码中对 FileInfo 类的成员变量进行访问, 并将文件信息打印输出。下面是代码执行后的输出结果:

```

文件名:      c-sharp.txt
文件扩展名:  .txt
文件全称:    C:\c-sharp.txt
所在目录:    C:\
文件大小:    2020 bytes
文件属性:    ReadOnly, Hidden, Archive

```

FileInfo 类的成员方法的使用与成员变量的使用类似, 也是对调用对象所代表的文件进行操作。

20.3 对文件夹进行操作

本节将学习如何对指定的文件夹进行操作, 例如文件夹的复制、移动、删除和重命名。对文件夹的相关操作, 主要是通过使用 System.IO.Directory 类和 System.IO.DirectoryInfo 类来进行。

20.3.1 C#对文件夹操作的类

与 File 类和 FileInfo 类相似, Directory 类与 DirectoryInfo 类可以实现对文件夹的各种操作。同样, System.IO.Directory 类中包含对文件夹操作的静态方法, 而 System.IO.DirectoryInfo 类的对象中包含文件夹的各种成员变量和方法。

20.3.2 用 System.IO.Directory 类的静态方法操作文件夹

如表 20.5 所示, 描述了 System.IO.Directory 类中可以对文件夹进行操作的静态成员方法。

表 20.5 System.IO.Directory类中的静态成员方法

成 员 方 法	描 述
CreateDirectory()	创建指定的文件夹
Delete()	删除指定的文件夹
Exists()	检验指定的文件夹是否存在, 返回值为boolean类型
GetCreationTime()	返回包含文件夹创建时间信息的DateTime对象
GetDirectories()	返回目录中包含的子文件夹名字的字符串数组

续表

成员方法	描述
GetFiles()	返回指定文件夹下所包含的文件名字的字符串数
GetFileSystemEntries()	返回指定目录包含的子文件夹和文件名字的字符串数组
GetParent()	返回包含指定文件夹的父目录信息的DirectoryInfo对象
Move()	将指定的目录（包括其中文件和子文件夹）移动到指定路径下

□ 对文件夹进行创建、检验和删除操作

下面的代码将演示如何使用表 20.5 所描述的方法。

```
private void directoryPrint()
{
    //打印文件信息
    System.Console.WriteLine("'C:\\test' 目录是否存在: "+
        Directory.Exists("C:\\test"));
    System.Console.WriteLine("创建'C:\\test' 目录: "+
        Directory.CreateDirectory("C:\\test"));
    System.Console.WriteLine("'C:\\test' 目录是否存在: "+
        Directory.Exists("C:\\test"));
    System.Console.WriteLine("'C:\\test' 目录的父目录是: "+
        Directory.GetParent("C:\\test"));
    System.Console.WriteLine("删除'C:\\test' 目录……");
    Directory.Delete("C:\\test", true);
    System.Console.WriteLine("'C:\\test' 目录是否存在: "+
        Directory.Exists("C:\\test"));
}
```

在上述代码中，依次调用了 Directory 类的静态方法来执行下列操作：

首先，检验是否存在 C:\test 文件夹，检验后会发现开始时不存在 C:\test 目录，之后对目录进行创建。再次检验是否存在 C:\test 文件夹，发现 C:\test 目录创建成功，并可以输出目录的父目录信息。最后，使用 Delete 方法删除创建的文件夹，传入参数 true，表示在删除文件夹时实行递归删除，删除时同时删除指定文件夹中的所有文件和子文件夹。最后进行检验，确认被删除的文件夹已不存在。

□ 获得一个文件夹中的内容

Directory 类共提供了 3 种方法来递归的获得文件夹中的内容，Directory.GetDirectories()返回的是指定文件夹的子文件夹列表，Directory.GetFiles()返回的是指定文件夹的文件列表，Directory.GetFileSystemEntries()返回的是指定文件夹的所有文件和子文件夹的列表。在下面的代码中，将获得系统中一个文件夹的内容的列表。

```
//定义 button1_Click() 事件处理函数
private void button1_Click(object sender, EventArgs e)
{
    //定义变量
    string folder = Environment.GetFolderPath(Environment.
        SpecialFolder.System);
    folder = folder.Substring(0, folder.LastIndexOf('\\'));
    string[] fileSystemEntries = System.IO.Directory.
        GetFileSystemEntries(folder);
    string[] files = System.IO.Directory.GetFiles(folder);
    string[] directories = System.IO.Directory.GetDirectories(folder);
    //输出文件信息
```



```

this.listBox1.Items.Add("文件夹的地址"+folder);
this.listBox1.Items.Add("");
this.listBox1.Items.Add("文件夹里的文件实体有");
//使用 foreach 循环遍历 fileSystemEntries 内数据
foreach (string fileSystemEntry in fileSystemEntries)
{
    this.listBox1.Items.Add("\t" + fileSystemEntry);
}
this.listBox1.Items.Add("");
this.listBox1.Items.Add("文件夹里的文件有");
foreach (string file in files)
{
    this.listBox1.Items.Add("\t" + file);
}
this.listBox1.Items.Add("");
this.listBox1.Items.Add("文件夹里的文件夹有");
foreach (string directory in directories)
{
    this.listBox1.Items.Add("\t" + directory);
}
}

```

运行结果如图 20.3 所示。



图 20.3 文件夹内容查询运行结果

技巧：在上面的事件处理程序中，Directory.GetFileSystemEntries(folder)方法返回了一个字符串序列，每一个字符串展示了文件夹里的文件实体，可通过 foreach 语句遍历字符串序列来获得每一个单独的文件实体。

20.3.3 用 System.IO.DirectoryInfo 类的方法操作文件夹

DirectoryInfo 类也是用来对文件夹执行各种操作，与 Directory 类不同的是，在使用 DirectoryInfo 类时需要创建 DirectoryInfo 类的对象。如表 20.6 所示，描述了 DirectoryInfo 类中的常用方法和属性。

表 20.6 DirectoryInfo 类的常用方法和属性

成员方法	描述
Extention	表示文件夹的类型（扩展名）
FullName	表示文件夹的全路径名
Name	表示文件夹的名字

续表

成员方法	描述
Parent	表示指定文件夹的父级文件夹的全路径
Create()	创建文件夹
Delete()	删除指定文件夹
Exists()	指示是否存在指定的文件夹
GetDirectories()	返回指定目录包含的所有子文件夹的DirectoryInfo类型的实例数组
GetFiles()	返回指定目录包含的所有文件的FileInfo类型的实例数组
GetFileSystemInfos()	返回指定目录包含的所有的子文件夹和文件的FileSystemInfo实例
MoveTo()	移动这个目录及里面的所有文件至指定路径
Refresh()	刷新这个DirectoryInfo类的实例

在下面的代码示例中,将会更改前一个代码段,使用 DirectoryInfo 类代替 Directory 类,改变的代码如下:

```
//定义 button1_Click() 事件处理函数
private void button1_Click(object sender, EventArgs e)
{
    //定义变量
    string folder = Environment.GetFolderPath(Environment.SpecialFolder.System);
    folder = folder.Substring(0, folder.LastIndexOf('\\'));
    System.IO.DirectoryInfo folderInfo = new System.IO.
    DirectoryInfo(folder);
    FileSystemInfo[] fileSystemInfos = folderInfo.GetFileSystemInfos();
    FileInfo[] fileInfos = folderInfo.GetFiles();
    DirectoryInfo[] directoryInfos = folderInfo.GetDirectories();
    //输出文件信息
    this.listBox1.Items.Add("文件夹的地址"+folder);
    this.listBox1.Items.Add("");
    this.listBox1.Items.Add("文件夹里的文件实体有");
    //使用 foreach 循环遍历 fileSystemInfos 内数据
    foreach (FileSystemInfo fileInfo in fileSystemInfos)
    {
        this.listBox1.Items.Add("\t" + fileInfo.FullName);
    }
    this.listBox1.Items.Add("");
    this.listBox1.Items.Add("文件夹里的文件有");
    foreach (FileInfo fileInfo in fileInfos)
    {
        this.listBox1.Items.Add("\t" + fileInfo.FullName);
    }
    this.listBox1.Items.Add("");
    this.listBox1.Items.Add("文件夹里的文件夹有");
    foreach (DirectoryInfo directoryInfo in directoryInfos)
    {
        this.listBox1.Items.Add("\t" + directoryInfo.FullName);
    }
}
```

运行结果如图 20.4 所示。

经过比较可以发现,两段代码的运行结果完全相同。而代码中的区别在于将 Directory 类中的静态方法改为了 DirectoryInfo 类中的方法和属性。


 **技巧：**在 .NET 中可以很容易地获取到文件系统和应用程序环境的信息。但是在应用程序中，需要对文件及文件夹进行谨慎操作。在本节中，并没有就文件及文件夹操作中发生的错误进行异常处理，但在实际编程中，应对文件进行操作的代码用 try...catch 语句包围起来。



图 20.4 文件夹内容查询运行结果

20.4 流文件概述

在编程过程中，总是存在着各种各样的流（Stream）。对于流的解释并没有统一的定论，主要集中在以下两种说法：

- 流是用于对数据文件进行读写数据的对象。
- 流是一个抽象的比特序列。

形象的解释，流就像是连接两个终端的桥梁，一端连接着数据源，例如文件、存储中的数据或者另一台 PC 上的数据，另一端连接着对数据进行读写操作的类。

20.4.1 了解流

由于存在着多种数据源，因此数据也可以从多种介质文件中读取，例如磁盘、远程计算机、计算机内存或者一些输入输出设备。但是为了开发的简洁性，需要一个简单的接口，仅利用简单的方法从不同的数据源读取数据，例如 Read()方法和 Write()方法。

20.4.2 Stream 类的常用方法和属性

在前面的各章节中，有很多地方已经用到了 Console.WriteLine()方法和 Console.ReadLine()方法。实际上，.NET 中的 Console 类可以将输入流、输出流和错误流显示在控制台窗体上。通过调用 Console 类的 Write()方法，可以将数据输出到控制台窗体上。

System.Stream 类是其他流文件类的抽象类。其他流都是通过对该类的属性和方法进行重写而获得的。该类的常用属性和方法如表 20.7 所示。

表 20.7 System.Stream类的常用方法和属性

成 员 方 法	描 述
CanRead	一个boolean类型的属性, 指示流是否能从数据源中读信息
CanWrite	一个boolean类型的属性, 指示流是否能向数据源中写信息
Length	当前流中的字节长度
Position	流的当前位置
Close()	关闭流
Flush()	将流中的数据全部写入数据源
Seek()	设置流文件的当前位置
Read()	从流的当前位置读入指定大小的比特流
ReadByte()	从流的当前位置读入一个字节
Write()	向流的当前位置写入指定大小的比特流
WriteByte()	向流的当前位置写入一个字节

20.5 使用流对文件进行读写

在.NET 类库中, 有很多类可以对文件进行读写。开发人员可以使用 System.IO.FileStream 类对文件进行读写操作, 可以使用 System.IO.BinaryReader 类和 System.IO.BinaryWriter 类对一些原始数据进行读取或写入二进制流, 也可以使用 System.IO.StreamReader 类和 System.IO.StreamWriter 类来读写文本文件。

20.5.1 使用 System.IO.FileStream 类进行文件读写

System.IO.FileStream 类继承自 System.IO.Stream 类, 它重写了 Stream 类中的所有关于文件操作的抽象方法。在使用这些流操作前, 首先需要使用如下语句在程序的顶部添加 System.IO 命名空间。

```
using System.IO;
```

在编程过程中, 可以使用很多方法实例化 FileStream 类, 例如使用 System.IO.File 类的 File.Open()方法、File.OpenRead()方法和 File.OpenWrite()方法。

```
FileStream fs = File.Open("C:\\c-sharp.txt", FileMode.Open);
```

同样, 也可以使用 System.IO.FileInfo 类的 Open()方法、OpenRead()方法和 OpenWrite()方法。

```
FileInfo fileInfo = new FileInfo("C:\\c-sharp.txt");
FileStream fs = File.Open(fileInfo.FullName, FileMode.Open);
```

也可以使用 FileStream 类的构造函数来实例化一个 FileStream 类, 当创建文件流时, 可能需要定义下面 4 个参数。

- ❑ 文件的名称和路径。
- ❑ FileMode, FileMode 的枚举类型参数定义了系统如何打开文件, FileMode 的成员主要枚举值有 FileMode.Open、FileMode.OpenOrCreate、FileMode.Append 和

FileMode.Create。如果 FileMode 枚举类型值是 FileMode.Open，那么系统将尝试打开现有的文件；如果 FileMode 枚举类型值是 FileMode.OpenOrCreate，那么系统将尝试打开现有的文件，如果这个文件不存在，则创建这个文件；如果 FileMode 枚举类型值是 FileMode.Create，那么系统将创建这个文件；如果 FileMode 枚举类型值是 FileMode.Append，那么系统会将新的数据写在现有的文件的结尾处。

- FileAccess，FileAccess 的枚举类型参数定义了对文件进行访问时所允许的操作，常用的值主要包括 FileAccess.Read、FileAccess.Write 和 FileAccess.ReadWrite。如果在参数中 FileAccess 的枚举类型值是 FileAccess.Read，则流可以对文件进行读操作；如果在参数中 FileAccess 的枚举类型值是 FileAccess.Write，则流可以对文件进行写操作；如果在参数中 FileAccess 的枚举类型值是 FileAccess.ReadWrite，则流可以对文件进行读写操作。
- FileShare，FileShare 的枚举类型参数定义了在文件共享时的选项，常用的值主要包括 FileShare.None、FileShare.Read、FileShare.Write 和 FileShare.ReadWrite。如果在参数中 FileShare 的枚举类型值是 FileShare.None，则除了这个流以外，没有流可以对文件进行操作；如果在参数中 FileShare 的枚举类型值是 FileShare.Read，则其他流只可以对文件进行读操作；如果在参数中 FileShare 的枚举类型值是 FileShare.Write，则其他流只可以对文件进行写操作；如果在参数中 FileShare 的枚举类型值是 FileShare.ReadWrite，则其他流可以对文件进行读写操作。

20.5.2 用 System.IO.FileStream 类打开文本文件

本节将通过一个具体的 Windows 应用程序来讲解 System.IO.FileStream 类的使用方法。

在演示示例中，将创建一个 Windows 应用程序，这个应用程序从一个名为 c-sharp.txt 的文本文件中读取文本内容，并将其内容显示在 Windows 窗体的文本框中，同时允许用户对文本框中的内容进行修改并将它保存回原文本文件中。首先在窗体设置器中创建 Windows 窗体，然后按照图 20.5 所示界面对窗体进行设计。

为了使程序能够正确运行，读者需要保证在 C 盘中有一个名为 c-sharp.txt 的文档。并在“读文件”按钮的 click 事件中添加如下代码：

```
//定义 btnRead_Click() 事件处理函数
private void btnRead_Click(object sender, EventArgs e)
{
    //定义变量
    string fileName = "C:\\c-sharp.txt";
    this.label1.Text = "文件路径"+fileName;
    //打开已存在的文件
    FileStream fs = new FileStream(fileName, FileMode.Open, FileAccess.Read,
    FileShare.None);
    //创建一个字符数组
    byte[] byteText = new byte[fs.Length];
    //将文件中的内容读入数组
    fs.Read(byteText, 0, byteText.Length);
    //将字符数组转化为字符串并显示在文本框中
```

```

this.textBox1.Text = System.Text.Encoding.ASCII.GetString(byteText);
//关闭类
fs.Close();
}

```



图 20.5 Windows 窗体外观

在上面的代码中，已经使用文件流以只读的方式打开了一个存在的文件，并且当此应用程序使用 `c-sharp.txt` 文件时，不允许其他流再对此文件进行访问。然后使用下面的代码创建了一个与流文件大小相等的比特字符数组，用以存储文件中的数据。

```
fs.Read(byteText, 0, byteText.Length);
```

这里需要特别指出的是，如果需要将文件的内容读入比特数组，那么文件流应该从数组的索引号为 0 的位置开始填充。读入这些文件内容后，需要将这个字符数组转化为字符串以使其可以显示在文本框中。最后，程序将关闭这个文件流，以使其他文件流可以访问这个文件。

注意：在对文件的操作结束时，一定要关闭对此文件进行操作的文件流，以使后面的操作能够正常进行。

同样地，“存储”按钮的 `click` 事件也可以如此处理，代码如下：

```

//定义 btnSave_Click() 事件处理函数
private void (object sender, EventArgs e)
{
    //定义变量
    string fileName = "C:\\c-sharp.txt";
    //打开已存在的文件
    FileStream fs = new FileStream(fileName, FileMode.Open,
    FileAccess.Write, FileShare.None);
    //创建一个字符数组
    byte[] byteText =
    System.Text.Encoding.ASCII.GetBytes(this.textBox1.Text);
    //将数组中的内容写入文件
    fs.Write(byteText, 0, byteText.Length);
    //关闭类
    fs.Close();
}


```

在上面的代码中，将文本框中内容转化成一个比特数组，并通过使用 `FileStream` 的 `Write()` 方法将其写回到文件中。

20.5.3 用 BinaryReader 和 BinaryWriter 类读写原始文件

使用 `FileStream` 类的局限性在于开发人员只能读写文件的比特流，在使用此类时，开发人员需要将其他的数据类型转化为比特流后才能对文件进行读写。

针对这一局限性，.NET 类库提供了两个类来对文件的原始数据进行读写，这两个类分别是 `BinaryReader` 类和 `BinaryWriter` 类。开发人员可以使用 `System.IO.BinaryReader` 类来从一个文件中读取原始数据，使用 `System.IO.BinaryWriter` 类来对一个原始文件执行写操作。

 **说明：**对于 `BinaryReader` 类和 `BinaryWriter` 类最重要的一点是，这两个类的构造函数都需要传入一个文件流。

在下面的示例程序中，将重点讲解 `BinaryReader` 类和 `BinaryWriter` 类的应用。

在窗体设计器中创建如图 20.5 所示的外观的窗体，并将文本框设置为只读类型。在该示例程序中，程序启动时，“读文件”按钮是不可用的，用户需要首先单击“存储”按钮来存储文件，然后“读文件”按钮才可用，可通过单击此按钮将文件读入到文本框中。在“存储”按钮的 `click` 事件中添加如下代码：

```
//定义 btnSave_Click() 事件处理函数
private void btnSave_Click(object sender, EventArgs e)
{
    //定义变量
    string fileName = "C:\\c-sharp.txt";
    //打开已存在的文件
    FileStream fs = new FileStream(fileName, FileMode.Open,
        FileAccess.Write, FileShare.None);
    //在这个文件流上创建 BinaryWriter
    BinaryWriter writer = new BinaryWriter(fs);
    //向文件里写入原始数据
    writer.Write("这是一个 System.IO 类的测试程序\r\n");
    writer.Write("这是一个 FileStream 类的测试程序\r\n");
    writer.Write("这是一个 BinaryReader 类的测试程序\r\n");
    writer.Write("这是一个 BinaryWriter 类的测试程序\r\n");
    //关闭类
    writer.Close();
    fs.Close();
    //设置标志值
    btnRead.Enabled = true;
}
```

在上面的代码中，生成了一个文件流并通过它创建了一个 `BinaryWriter` 类，然后通过使用 `BinaryWriter` 类的 `Write()` 方法向此流中写入原始数据，最后关闭两个流文件，并使“读文件”按钮可用。

在“读文件”按钮的 `click` 事件中添加如下代码：

```
//定义 btnRead_Click() 事件处理函数
private void btnRead_Click(object sender, EventArgs e)
{
    //定义变量
    string fileName = "C:\\c-sharp.txt";
```

```

this.label1.Text = "文件路径"+fileName;
//打开已存在的文件
FileStream fs = new FileStream(fileName, FileMode.Open, FileAccess.Read,
FileShare.None);
//在这个文件上创建 BinaryReader
BinaryReader reader = new BinaryReader(fs);
//从流中读数据
string str1 = reader.ReadString();
string str2 = reader.ReadString();
string str3 = reader.ReadString();
string str4 = reader.ReadString();
//将数据写入文本框
this.textBox1.Text=str1+str2 +str3+str4;
//关闭流
reader.Close();
fs.Close();
}

```

在上面的代码中，使用 `FileStream` 类生成了 `BinaryReader` 类的对象，并用其读出写入到文件里的数据。将数据全部读出后，程序将读出的数据整合成一个字符串，并使它显示到文本框中。在程序的结尾处，关闭两个流。程序的运行结果如图 20.6 所示。



图 20.6 程序运行结果

20.5.4 用 StreamReader 和 StreamWriter 类读写文本文件

通过使用 `StreamReader` 类和 `StreamWriter` 类可以实现对文本文件的读写，这两个类提供了很多有效的方法使对文件的读写变得非常容易，例如 `ReadLine()`方法和 `ReadToEnd()`方法。`StreamReader` 和 `StreamWriter` 类的使用方法与 `BinaryReader` 和 `BinaryWriter` 类非常相似，在此处不再赘述。

技巧：`StreamReader` 和 `StreamWriter` 的实例化对象可以使用不同的文本编码方式（例如 ASCII 和 Unicode）来读写文件。

20.6 序列化和反序列化

序列化是将对象写入流的过程，而反序列化则是从流中读取对象的过程。在本章前面

的部分，只是提到了如何使用 `BinaryReader` 类和 `BinaryWriter` 类从流中读写原始数据，而如何从流中读取其他类型的数据（例如，开发人员定义的类型）将在本节中进行介绍。

20.6.1 什么是对象序列化

序列化一个对象是为了可以存储这个对象当前的状态，而这个状态主要是由这个对象中的变量确定的。也就是说，序列化一个对象也就是将这个对象的所有成员变量的值写入流中，而这个对象的方法和静态成员并不写入流中。

因此，可以通过将开发人员自己定义的类型实例化对象中的成员变量依次写入流中来对此对象实现序列化。当反序列化时，则需要按照它们被写入的顺序来读取所有的成员变量。但是，通过这种方法实现的序列化和反序列化也存在着一定的局限性。

- ❑ 将开发人员所定义的类的实例化对象的所有成员变量全部写入流中是一个非常乏味的工作，当这个类中的成员变量很多时，这一过程的工作量将变得很大。
- ❑ 在序列化的时候，并没有一个标准的可遵循的顺序，因此，如果其他开发人员需要反序列化这个流时，必须清楚地了解此流序列化时的顺序，并遵从此顺序进行反序列化。

为了避免以上两个问题，在 .NET 框架中，开发人员可以通过调用 `Serialize()` 方法和 `DeSerialize()` 方法来处理对象的序列化操作。但是在有的应用程序中，并不是希望所有的类或类中所有的成员变量都可以被序列化，可以通过对类或成员变量设置相关的属性来标注此类或此成员变量是否允许被序列化。当开发人员所定义的类可以被序列化时，可以在此类的上方使用 `[Serializable]` 属性标注出此类是可序列化的。同样地，也可以使用 `[NonSerializable]` 属性标注出某成员变量是不可以被序列化的，以阻止 CLR 在序列化类时对此成员变量进行序列化。

20.6.2 用格式器描述被序列化对象

在对类的对象进行序列化时，可以使用格式器（formatter）来描述被序列化的对象的格式，格式器必须是符合一定标准的，并且在序列化和反序列化时都可使用一个统一的或兼容的格式器。在 .NET 中，一个格式器必须执行 `System.Runtime.Serialization.IFormatter` 接口，主要有两个常用的格式器，分别是二进制格式器（`System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`）和 SOAP（简单对象访问协议）格式器（`System.Runtime.Serialization.Formatters.Soap.SoapFormatter`）。

📌 说明：SOAP 是互联网上的一个标准协议，而二进制格式器则更适用于本地系统。

20.6.3 对象序列化使用示例一

本节将通过一个演示程序来讲解如何序列化一个用户定义类的对象。

在下面的例子中，将创建一个包含有一个序列化类的简单的控制台应用程序，这个应

用程序将一个对象序列化至一个文件，然后通过反序列化将此文件写入到另一个对象中。程序的具体代码如下：

```
//声明使用的命名空间
using System;
//包含 FileStream 类
using System.IO;
//包含 BinaryFormatter 类
using System.Runtime.Serialization.Formatters.Binary;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _20._6._3
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //创建 Add 类的对象
            Add add = new Add(1, 2);
            //打开已存在的文件
            FileStream fs = new FileStream("C:\\C-sharp.txt",
            FileMode.Create);
            BinaryFormatter binFormatter = new BinaryFormatter();
            //对对象进行序列化
            Console.WriteLine("对对象进行序列化...");
            binFormatter.Serialize(fs, add);
            //转至文件起始处
            fs.Position = 0;
            //对文件进行反序列化
            Console.WriteLine("对文件进行反序列化...");
            Add sum = (Add)binFormatter.Deserialize(fs);
            int result = sum.Apply();
            Console.WriteLine("1+2=" + result.ToString());
        }
    }
    //定义 Add 类
    [Serializable]
    class Add
    {
        //私有成员变量
        private int num1;
        private int num2;
        private int sum;
        //定义 Add 类的构造函数
        public Add()
        {
        }
        public Add(int num1, int num2)
        {
            this.num1 = num1;
            this.num2 = num2;
        }
        //定义 Apply() 方法
        public int Apply()
        {
            sum = num1 + num2;
        }
    }
}
```



```

        return sum;
    }
    //定义 Result 属性
    public int Result
    {
        get
        {
            return sum;
        }
    }
}

```

在上述代码中，Add 是一个开发人员自行定义的类，这个类非常简单，只包含 3 个私有成员。在上面的例子中，在 Add 类前面标记了[Serializable]属性，指明此类可以进行序列化，同时使用如下语句引用了相关的命名空间：

```

using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

```

在 Main()方法中，程序创建了 Add 类的实例，然后创建了一个文件流。并且使用 BinaryFormatter 类来将这个对象序列化至文件中。下面的代码就是将类序列化为文件时所需要做的所有工作：

```

FileStream fs = new FileStream("C:\\C-sharp.txt", FileMode.Create);
BinaryFormatter binFormatter = new BinaryFormatter();
Console.WriteLine("对对象进行序列化");
binFormatter.Serialize(fs, add);

```

而当反序列化时，所需要的代码与序列化很相似，但却需要将文件指针设置到文件的起始处。如下面的代码所示。

```

fs.Position = 0;

```

设置完文件指针后可以使用如下代码对文件进行反序列化：

```

Console.WriteLine("对文件进行反序列化");
Add sum = (Add)binFormatter.Deserialize(fs);

```

BinaryFormatter 类的 Deserialize()方法将文件流作为它的参数，从此流中读取对象的信息并且返回一个 System.Type 类型的对象，并将其传给需要的类。一旦程序得到了这个对象，则将会调用类的 Apply()方法来计算类中私有变量的和并将其打印到控制台上，如下面的代码所示。

```

int result = sum.Apply();
Console.WriteLine("1+2="+result.ToString());


```

对程序进行编译后，得到下面的结果：

```

对对象进行序列化...
对文件进行反序列化...
1+2=3
按任意键继续

```

 注意：如果将上述程序中 Add 类的 Serializable 属性注释掉，则在调用 BinaryFormatter 类的 Serialize()方法时会抛出异常。

20.6.4 对象序列化使用示例二

上文提到,开发人员可以通过对类中的成员变量的属性进行设置,使 CLR 在序列化对象时不能序列化某个变量。可以通过将上文的代码更改为下面的形式来实现这种需求:

```
//声明使用的命名空间
using System;
//包含 FileStream 类
using System.IO;
//包含 BinaryFormatter 类
using System.Runtime.Serialization.Formatters.Binary;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace _20._6._4
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //创建 Add 类的对象
            Add add = new Add(1, 2);
            //调用 Apply() 方法
            add.Apply();
            Console.WriteLine("运行结果是" + add.Result.ToString());
            //打开已存在的文件
            FileStream fs = new FileStream("C:\\C-sharp.txt",
            FileMode.Create);
            BinaryFormatter binFormatter = new BinaryFormatter();
            //对对象进行序列化
            Console.WriteLine("");
            binFormatter.Serialize(fs, add);
            //转至文件起始处
            fs.Position = 0;
            //对文件进行反序列化
            Console.WriteLine("对文件进行反序列化");
            Add sum = (Add)binFormatter.Deserialize(fs);
            int result = sum.Apply();
            Console.WriteLine("运行结果是" + add.Result.ToString());
            Console.WriteLine("1+2=" + result.ToString());
        }
    }
    //定义 Add 类
    [Serializable]
    class Add
    {
        //私有成员变量
        private int num1;
        private int num2;
        [NonSerialized]
        private int sum;
        //定义 Add 类的构造函数
        public Add()
        {
```



```

    }
    public Add(int num1, int num2)
    {
        this.num1 = num1;
        this.num2 = num2;
    }
    //定义 Apply() 方法
    public int Apply()
    {
        sum = num1 + num2;
        return sum;
    }
    //定义 Result 属性
    public int Result
    {
        get
        {
            return sum;
        }
    }
}

```

在上述代码中，对 Add 类的 sum 成员变量添加了[NonSerialized]属性。在 Main()方法中，首先创建了 Add 类的实例，然后调用 Add()方法计算出结果并打印出来。具体如下面的代码所示。

```

Add add = new Add(1,2);
add .Apply();
Console.WriteLine("运行结果是"+add.Result.ToString());

```

第二步，程序序列化了这个类的对象并将它反序列化到另一个对象，这个过程与前一个例子没有区别。具体代码如下所示。

```

FileStream fs = new FileStream("C:\\C-sharp.txt", FileMode.Create);
BinaryFormatter binFormatter = new BinaryFormatter();
Console.WriteLine("对对象进行序列化");
binFormatter.Serialize(fs, add);
fs.Position = 0;
Console.WriteLine("对文件进行反序列化");
Add sum = (Add)binFormatter.Deserialize(fs);

```

反序列化后，程序通过以下代码首先打印出对象里的 Result 属性。

```

Console.WriteLine("运行结果是" + add.Result.ToString());

```

如果 Result 属性没有被序列化，那么上面代码的运行结果应该为 0。最后，打印出 Apply 的运行结果。

```

Console.WriteLine("1+2="+result.ToString());

```

如果 num1 和 num2 被序列化，那么上面代码的结果应该是 1 与 2 的和。对程序进行编译执行，得到如下的结果：

```

运行结果是 3
对对象进行序列化...
对文件进行反序列化...

```

```
运行结果是 0
1+2=3
按任意键继续
```

说明：未序列化的 `sum` 的值为 0，也就是说对象中的 `sum` 没有被序列化。而调用 `Apply()` 方法后所得到的结果为 3（1+2 的结果），说明在对象中的 `num1` 和 `num2` 变量被序列化了。

20.6.5 对象序列化使用示例三

有时，在程序中不希望将对象中的一个变量序列化，却希望在反序列化时可以对这个对象进行一些操作，而得到没有被序列化的变量，这时可以通过让类执行 `IDeserializationCallBack` 接口来实现。该接口只包含了一个方法，如下所示。

```
void OnDeserialization(Object sender)
```

在类执行 `IDeserializationCallBack` 接口的时候，这个方法总是被调用。更改前面的应用程序，使 `result` 值即使没有被序列化也不会改变，更改后的代码如下所示。

```
//声明使用的命名空间
using System;
//包含 FileStream 类
using System.IO;
//包含 BinaryFormatter 类
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Collections.Generic;
using System.Text;
namespace _20._6._5
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //创建 Add 类的对象
            Add add = new Add(1,2);
            add.Apply();
            Console.WriteLine("运行结果是"+add.Result.ToString());
            //打开文件
            FileStream fs = new
            FileStream("C:\\C-sharp.txt",FileMode.Create);
            BinaryFormatter binFormatter = new BinaryFormatter();
            //对对象进行序列化
            Console.WriteLine("对对象进行序列化");
            binFormatter.Serialize(fs,add);
            //转至文件起始处
            fs.Position = 0;
            //对文件进行反序列化
            Console.WriteLine("对文件进行反序列化");
            Add sum = (Add)binFormatter.Deserialize(fs);
            int result = sum.Apply();
            Console.WriteLine("运行结果是" + add.Result.ToString());
        }
    }
}
```



```

    }
}
//定义继承 IDeserializationCallBack 类的 Add 类
[Serializable]
class Add:IDeserializationCallBack
{
    //定义私有成员变量
    private int num1;
    private int num2;
    [NonSerialized]
    private int sum;
    //定义 Add 类的构造函数
    public Add()
    {
    }
    public Add(int num1, int num2)
    {
        this.num1 = num1;
        this.num2 = num2;
    }
    //定义 Apply() 方法
    public int Apply()
    {
        sum = num1 + num2;
        return sum;
    }
    //定义 Result 属性
    public int Result
    {
        get
        {
            return sum;
        }
    }
    //定义 OnDeserialization() 方法
    public void OnDeserialization(Object sender)
    {
        sum=num1+num2;
    }
}
}

```

在上述方法中，Add 类继承了 IDeserializationCallBack 接口，并且提供了 OnDeserialization()方法的定义，使其可以计算出 num1 与 num2 的和，并将结果存储在私有成员变量 sum 里。

```

public void OnDeserialization(Object sender)
{
    sum=num1+num2;
}

```

上面例子的 Main()方法与前一个例子非常类似，但是在反序列化时，由于 OnDeserialization()方法被调用，程序将得不到 sum 的初值 0。编译并执行以上程序，将得到以下的结果：

```

运行结果是 3
对对象进行序列化...
对文件进行反序列化...

```

运行结果是 3
按任意键继续

20.7 异步读取数据

在前面的各节中，主要讨论了如何实现同步地对流进行读写。而在本节中，将重点讲述如何对流实现异步读写。

20.7.1 什么是异步读取数据

在前面所编写对流进行读写的程序中，都是通过调用 `Read()` 和 `Write()` 方法来执行对流的读写，例如，可以使用 `Read()` 方法将流中的内容读进一个数组中，具体代码如下所示。

```
byte[] byteText = new byte[fs.Length];  
fs.Read(byteText, 0, byteText.Length);  
DoOthers();
```

在上面的代码中，当调用 `Read()` 方法时，程序（或当前的线程）将进入阻塞状态，直到代码被读取到一个指定的数组中，`DoOthers()` 方法才会被调用，这就是一个同步读的过程。而在异步读写过程中，程序只需要通过调用 `System.IO.Stream` 类的 `BeginRead()` 方法来 `BeginWrite()` 方法来启动读或写操作，而不必进入阻塞状态。如果同时调用了 `BeginRead()` 方法和 `BeginWrite()` 方法，那么这两个任务将会同时开始。

在使用 `BeginRead()` 和 `BeginWrite()` 方法时将会获得一个 `System.AsyncCallback` 类的代理，该代理在系统中的定义如下所示。

```
public delegate void AsyncCallback(IAsyncResult ar)
```

从上面的定义可以看出，`AsyncCallback` 传入一个 `IAsyncResult` 类型的参数，而没有返回值，此代理能够被用于任何方法。`IAsyncResult` 类型的参数是用来为异步操作提供信息，在大多数时候，不需要对这个参数进行更改。在下面的代码中，演示了如何引用一个 `AsyncCallback` 代理：

```
public void OnWriteCompletion(IAsyncResult ar)  
{  
    Console.WriteLine("写操作完成");  
}
```

20.7.2 异步读取数据使用示例

本节将通过一个演示程序来讲解如何对数据实现异步读取。

在下例中，将创建一个简单的控制台程序来实现对流的异步读写，程序的代码如下：

```
//声明使用的命名空间  
using System;  
using System.IO;  
using System.Threading;  
using System.Collections.Generic;
```



```

using System.Text;
namespace _20._7._2
{
    //定义 Program 类
    class Program
    {
        static void Main(string[] args)
        {
            //打开文件
            FileStream fs = new FileStream("C:\\C-sharp.txt", FileMode.Open);
            byte[] data = new byte[fs.Length];
            //读取文件数据
            Console.WriteLine("读文件...");
            fs.Read(data, 0, data.Length);
            //向文件写内容
            fs.Position = 0;
            AsyncCallback callBack = new AsyncCallback(OnWriteCompletion);
            fs.BeginWrite(data, 0, data.Length, callBack, null);
            Console.WriteLine("执行写操作...");
            //循环计数
            for(int i=0; i<10;i++)
            {
                Console.WriteLine("计数器执行置: {0}", i);
            }
            //关闭文件
            fs.Close();
        }
        //定义 OnWriteCompletion() 方法
        static void OnWriteCompletion(IAsyncResult ar)
        {
            Console.WriteLine("写操作完成");
        }
    }
}

```

在上面的代码段中, Main()方法定义了一个 AsyncCallback 类型的代理的实例 callBack, 并使其引用了 OnWriteCompletion()方法。

在程序的开始处创建了一个文件流, 使用该流读取数据至 fileData 数组中, 然后将文件流的指针移至文件的开始处。使用 BeginWrite()方法开始异步写文件, 并将比特数组传递给 callBack 代理的 OnWriteCompletion()方法。具体代码如下:

```

AsyncCallback callBack = new AsyncCallback(OnWriteCompletion);
fs.BeginWrite(data, 0, data.Length, callBack, null);

```


对代码进行编译并执行, 所得到的结果如下:

```

读文件...
执行写操作...
计数器执行置: 0
计数器执行置: 1
计数器执行置: 2
计数器执行置: 3
计数器执行置: 4
写操作完成
计数器执行置: 5
计数器执行置: 6

```

```
计数器执行置: 7  
计数器执行置: 8  
计数器执行置: 9
```

说明: 通过上述结果可以看出, 程序中 Main()方法的执行没有因写操作的进行而被阻断, 而是继续打印着迭代的数字, 写操作执行完成时, 系统将会打印出相关信息, 而循环还在继续进行。

20.8 本章总结

在本章中, 首先介绍了如何在程序中获得读取文件系统环境的相关信息, 如计算机名字、当前用户的用户名称、各种文件夹的标准路径等。然后介绍了在.NET 中如何对文件或文件夹进行操作, 如移动、复制、删除等。最后, 介绍了.NET 类库中的各种流, 以及如何使用这些流对文件进行操作, 如读写操作、序列化操作等。

20.9 实战练习

1. 在 Visual Studio 2010 中新建一个控制台应用程序, 编写代码将当前计算机使用的操作系统版本、Windows 系统文件夹名称、有哪几个驱动器等信息保存到文本文件 MyComputer.txt 中。

2. 在 Visual Studio 2010 中新建一个 Window 窗体应用程序, 在窗体中添加一个文本框和一个按钮, 为按钮编写代码, 将 C 盘所有文件夹名称、大小、创建日期显示在文本框中。

3. 在 Visual Studio 2010 中新建一个 Window 窗体应用程序, 在窗体中添加一个文本框和一个按钮, 为按钮编写代码, 将第 1 题生成的 MyComputer.txt 文件的内容读出并显示在文本框中。

4. 在 Visual Studio 2010 中新建一个 Window 窗体应用程序, 创建一个保存员工信息的类 Emp, 设置为该类允许序列化。然后在窗体中添加多个文本框和两个按钮, 文本框用来接收用户输入员工的信息, 一个按钮为“保存员工信息”, 另一个按钮为“载入员工信息”。单击“保存员工信息”按钮时, 将输入到文本框中的员工信息保存在 Emp 类, 并进行数据序列化为 emp.dat 文件(二进制格式)。当单击“载入员工信息”按钮时, 从 emp.dat 文件中反序列化数据, 然后显示到窗体的文本框中。

第 21 章 可扩展标记语言


可扩展标记语言 (Extensible Markup Language, XML) 是一个通用的、平台独立的数据描述语言, 并且 XML 已经在计算机业界中广泛使用, 被各大公司所接受。万维网协会 (W3C) 也推出了一些 XML 的统一标准。

微软公司已经意识到了 XML 语言的重要性, 早在几年前, 就在其产品中添加了对 XML 技术的支持。在 .NET Framework 中, 也同样提供对 XML 技术的支持, 事实上, XML 技术已经应用在了 .NET Framework 中的很多方面, 例如配置文件、C# 源代码注释语言和 Web 服务等。

本章将主要讲述如何使用 XML, 以及 .NET Framework 对 XML 提供支持的技术。

21.1 认识 XML

简单地讲, XML 就是一种结构化的文本。由于文本可以显示在任何一种计算机平台上, 所以, 只要开发人员将数据描述成文本格式, 那么其他人员都可以对其中的数据进行阅读而不需要任何格式的转换。

说明: 正是由于文本可以显示在任何一种计算机平台上, 使生产商和供给者之间可以很好地共享数据。

下面是一个简单的 XML 文本的例子:

```
<?xml version="1.0" standalone="yes"?>
<Employees>
  <Employee EmployeeID="1">
    <FirstName>John</FirstName>
    <LastName>Smith</LastName>
    <Salaried>true</Salaried>
    <Wage>40000</Wage>
    <Active>>false</Active>
  </Employee>
</Employees>
```

从上例中可以看出, XML 文本中的数据被描述为元素和属性。元素是一个被尖括号包围的起始标签和结束标签(像 HTML 语言一样)。例如, 上例中, <Employees>是<Employees>元素的起始标签, 而</Employees>是<Employees>元素的结束标签。“/”字符指示了一个结束标签。<Employees>元素是 XML 文本中的第一个元素, 也就是通常所说的根节点元素。一个元素同样可以具有一些属性值, 在上面的例子中, EmployeeID 就是一个值为 1 的属性。

元素同样可以包含子元素。在上例中, <Employee>便是<Employees>的一个子元素。

可以通过对元素设置子元素来表达数据间的继承格式，一个 XML 文本应该是一个格式完善的文本，也就是说一个 XML 文本将具有一个根节点，每一个元素都具有起始和结束标签，每一个标签都是正确嵌套的。下面是一个不正确嵌套标签的例子：


```
<?xml version="1.0" standalone="yes"?>
<Employees>
  <Employee EmployeeID="1">
    <FirstName>John</FirstName>
    <LastName>Smith</LastName>
    <Salaried>true</Salaried>
    <Wage>40000</Wage>
    <Active>false
  </Employee>
  </Active>
</Employees>
```

在该例中，<Active>元素的结束标签添加到了<Employee>元素的结束标签后面。由于<Employee>元素是<Active>元素的父元素，所以<Active>元素的结束标签应该在<Employee>元素的结束标签前面。

开发人员可以写一个程序来读取一个格式正确的 XML 文档，但是由于一个格式正确的文档是以层级关系来展示数据的，那么就存在一些开发完整的通用的程序来读取 XML 数据，这种读取 XML 文本中数据的程序叫做 XML 解析器。在 .NET Framework 中，有很多种可用的 XML 文本解析器来访问 XML 数据。一个比较常用的 API（application programming interface）是文档对象模型（Document Object Model, DOM）。

21.1.1 文档对象模型的功能

XML DOM 是 W3C 提出的访问 XML 文档的标准 API。在 DOM API 中，将 XML 文档中的数据以树的形式展示出来。因为 XML 文档是分级结构的，因此可以创建一棵树，在树的节点和子节点上展示整个 XML 文档。通过从根节点开始的遍历，程序可以访问到树中的任意一个节点。

 **技巧：**DOM API 除了提供了遍历文档节点的方法外，还提供了一些其他的 service。读者可以在 www.w3.org/DOM 中查找到 XML DOM 的所有详尽的说明。

DOM API 的一些比较常用的功能有：

- ☐ 查找一个 XML 文档的根节点；
- ☐ 按照给出的标签名查找出相应的元素；
- ☐ 获得给出节点的所有子节点；
- ☐ 获得给出节点的父节点；
- ☐ 获得一个元素的标签名；
- ☐ 获得一个元素的相关联数据；
- ☐ 获得一个元素的属性；
- ☐ 获得一个属性的值；
- ☐ 在文本中添加、修改和删除元素；
- ☐ 在文本中添加、修改和删除属性；


□ 在文本中复制一个节点。

如上所示，DOM API 为开发人员提供了一系列丰富的功能，而 .NET Framework 也对 DOM API 提供了完整的支持。如何在 .NET 中使用 DOM API 进行开发，将在下面的章节中详细阐述。

DOM API 对于遍历和修改 XML 文档非常适用，但是对于如何在文档中查找任意元素和属性却给予了很少的支持。因此，对于此类需求，需要采用其他的 XML 技术，例如 XML Path Language (XPath)。

21.1.2 用 XPath 查询 XML 文档

XPath 是另一个由 W3C 提出的 XML 的相关技术的执行标准。XPath 是用来查询 XML 文档的，符合一定标准的节点列表的标准语言。一个 XPath 表达式将详细地指出位置和模式的双重匹配，也可以对 XPath 表达式提供布尔类型的操作符、字符串函数和运算符来创建对于 XML 文档的复杂查询。

 **技巧：** XPath 同样还能提供简单的数字操作，例如求和或者四舍五入。可以在 www.w3.org/TR/xpath 中查找到 XML XPath 的所有详细的说明。

XPath 的一些比较常用的功能有：

- 查找当前节点的全部子节点；
- 按照给定的标签确定节点，并查找出此节点的所有父节点；
- 按照给定的标签确定节点，并查找出此节点的最后一个子节点；
- 按照给定的属性确定节点，并查找出此节点的第 n 个子节点；
- 查找出拥有 <tag1> 标签或 <tag2> 标签的第一个子节点；
- 获得所有不具有指定属性的节点；
- 获得某数字元素的子节点的和；
- 获得子节点的数目。

如上所示，XPath 为开发人员提供了一系列丰富的功能，而 .NET Framework 也对 XPath 提供了完整的支持。如何在 .NET 中使用 DOM API 进行开发，将在下面的章节中给予详细的阐述。

21.1.3 了解可扩展样式表语言 XSL

根据 W3C 的解释，XSL（可扩展样式表语言）实际上是包含了 3 种不同的规则，它包括 XPath（XML 文档中的导航语言）、XSLT（转换 XML 文档的语言）和 XSL-FO（可扩展样式表语言—格式化对象）。其中，XSL-FO 是指以 XML 语法为基础的应用于 XML 文档上的一种格式形式，它可以影响文档的外观。XSL-FO 的开发还没有相对完善，所以在本章的后面内容将介绍 XPath 和 XSLT。

XSLT 是一种以 XML 为基础，可以转换 XML 文档的语言，XSLT 样式表主要用于将一个 XML 文档转化为另一个 XML 文档。可以使用 XSLT 样式表将 XML 文档转化为其他格式，例如 HTML、RTF 和 PDF。XSLT 同样也可以将 XML 文档转化为另一个 XML 文档。

例如，一个生产商使用一种格式结构创建了 XML 文档，而当它的供应商使用这个文档时，却希望此文档是另一种格式。那么，XSLT 样式表则可以将这个 XML 文档转化为供应商所希望的格式。

XPath 表达式在转换过程中可以用于 XSLT 样式表中，关于这点更详细的说明读者可以在 www.w3.org/TR/xslt 上找到帮助。

.NET 中已经添加了关于 XSLT 的支持，在后面的章节中，将通过具体的例子来讲解如何使用 XSLT 样式表来转换 XML 文档。

21.1.4 用 XML Schemas 设置数据元素和属性

正如前面所提到的，XML 对于在不同的组织间交换数据是一种非常好的格式。但是，如果组织间并没有对他们所要共享的数据规定一组详尽的格式时，那么 XML 并不能起到预想的作用，XML 文档里的数据也不能提供信息来定义 XML 文档的结构。

使用 DTD（文档类型定义）可以很好地描述 XML 文档的结构。DTD 详尽地给出了在 XML 文档中的元素和属性，也同样指出了元素所在的位置及其出现的次数。可以说，DTD 是一种传统的表达 XML 文档结构的方法。

如果一个 XML 文档有一个与其相关联的 DTD，那么解析器就可以读取相应的 DTD，并确定 XML 文档是否符合相应的 DTD。如果 XML 文档符合相应的 DTD，这就是一个有效的 XML 文档。如果一个 XML 文档是有效的，那么文档的接收者就会得到文档中的数据，然而并不是所有的 XML 都可以有效地解析。

DTD 有一个缺陷，它不能指出在 XML 文档中，与每一个数据相关联的元素和属性。例如，如果在一个 XML 文档中，有一个标签为 <OrderID> 的元素，那么 DTD 并不能指示出 Order ID 的类型是什么，是字符串、数字还是其他。

而 XML Schemas 却能解决 DTD 的这个缺憾。XML Schemas 不仅能提供 DTD 对于 XML 文档所提供的所有支持，而且，XML Schemas 也允许开发人员定义元素和属性的数据类型，以及数字类型的最大值和最小值、字符串类型的最大长度、枚举类型的相应枚举值。

同样，.NET 中也添加了关于 DTD 和 XML Schema 定义的支持。了解了 XML 的相关内容及其相关的技术后，在下面的章节中，将会重点讲述 .NET 对 XML 文档的支持。

21.1.5 .NET 中处理 XML 的相关类

在 .NET Framework 中，提供了很多可对 XML 文档执行操作的类，在后面的各章节中将对这些类进行运用。具体类的介绍如表 21.1 所示。

表 21.1 .NET 中的 XML 相关类

类 名	命 名 空 间	描 述
XmlReader	System.Xml	读取 XML 文档的抽象类
XmlTextReader	System.Xml	一个不含有有效性检查的 XML 文档解析器，可以访问具有只读特性的 XML 文档，能够从 XML 文档中快速地读取数据
XmlValidatingReader	System.Xml	一个含有有效性检查的 XmlReader 类的继承类，可以通过 DTD 和 XML Schema 对 XML 文档中的数据进行检查，并且读取 XML 文档中的数据

续表

类 名	命 名 空 间	描 述
XmlNodeReader	System.Xml	从一个XML DOM中读取数据
XmlWriter	System.Xml	对XML文档进行写操作的抽象类
XmlTextWriter	System.Xml	可以生成一个XML文档，提供了快速写XML文档的方法
XmlDocument	System.Xml	一个表示XML文档的类
XmlDataDocument	System.Xml	一个可执行的XML文档，允许数据通过XML DOM被访问
XPathDocument	System.Xml.XPath	一个适合使用XPath进行导航的XML文档
XPathNavigator	System.Xml.XPath	对于整个XML文档提供XPath导航
XmlSchema	System.Xml.Schema	一个XML Schema
XslTransform	System.Xml.Xsl	可执行的转换XML文档的语言

21.2 使用 XML DOM 进行编程

从 21.1.1 节的介绍中了解到 XML DOM API 主要用来创建、修改和横向遍历整个 XML 文档。在.NET Framework 中，System.Xml 命名空间中包含了很多可以对 XML DOM 文档进行操作的类。如表 21.2 所示，列举了在 System.Xml 命名空间中，程序员在关于 XML DOM 文档的开发时经常会使用到的类。

表 21.2 XML DOM相关类

类 名	描 述	类 名	描 述
XmlDocument	表示一个XML DOM文档的类	XmlElement	表示一个XML文档中的元素
XmlNode	表示一个XML文档中的节点	XmlAttribute	表示一个XML文档中的一个元素的属性
XmlNodeList	表示一个XML文档中节点的列表		

在下面的部分，将通过一个简单的例子来讲解.NET Framework 对 XML DOM 的支持。在这个例子中，将假设有一个小型的公司，其规模并不需要购进一个昂贵的商业用数据库来处理其所有的雇员信息。通常情况下，这种规模的公司只需要使用 Excel 提供的电子表格就可以记录并管理所有的雇员信息。

当然，还有另一种解决方案，就是将一个 XML 文档作为一个数据库来使用。可以通过使用.NET Framework 创建一个 XML 文档，并对其进行更新、保存和载入等操作。

如图 21.1 所示为本节例子中的相应程序的运行界面，在这个程序中，用户可以增加、删除和修改数据库中职员的信息。

当程序启动时，它将从当前文件夹中一个名为 Employees.xml 的包含员工信息的 XML 文件中读取数据。如果这个文件不存在，那么将开启一个新的空员工数据库，如果这个文件存在，它会将 XML 文档中的数据读到一个 XML DOM 文档里。程序将在机器内存中遍历整个 XML 文档。每当程序遇见一个 Employee 元素时，就会将这个雇员的姓名加载到窗体中雇员列表的下拉列表框中。

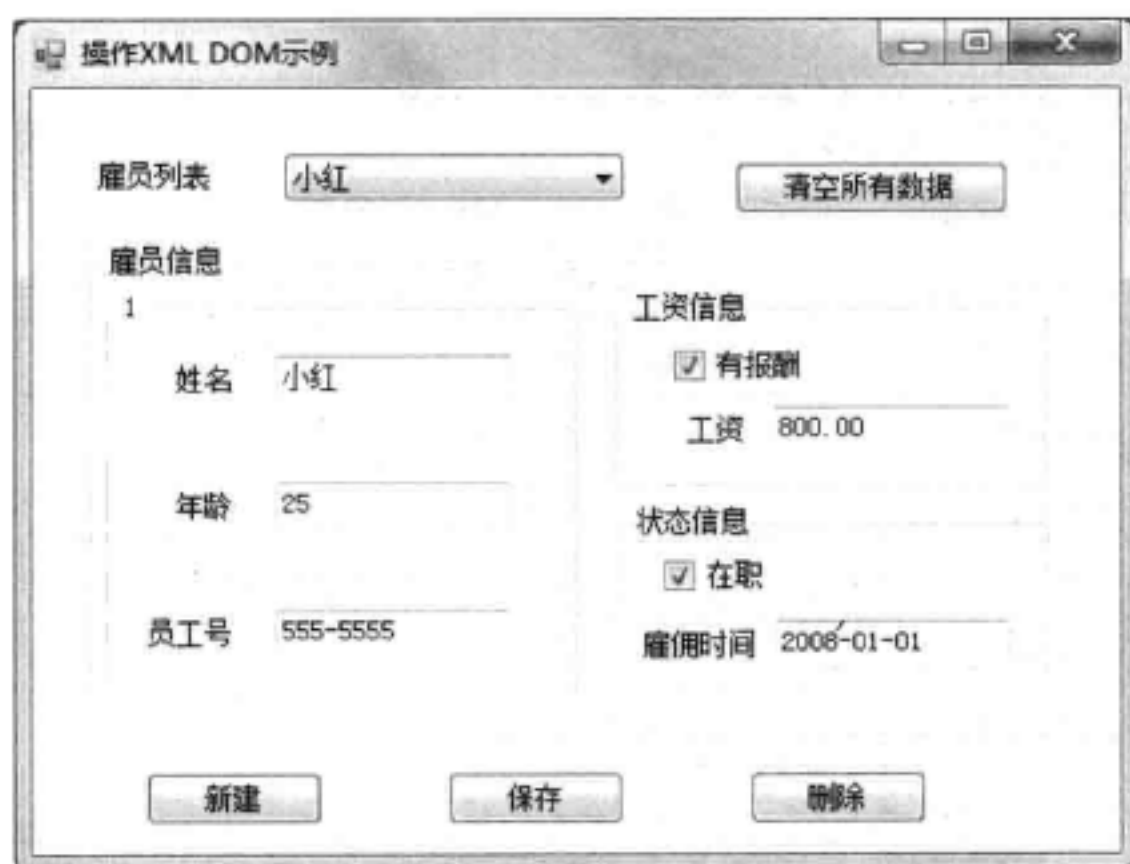


图 21.1 运行界面

当程序结束加载雇员信息后，用户就可以开始编辑用户信息。单击“新建”按钮，将清除“雇员信息”框图中的文本框中的内容，并允许用户在其中添加新的信息。单击“保存”按钮，将保存当前 XML 文档中所展示的用户信息，如果这个信息已经存在，那么它将更新相应的雇员信息；如果这条信息并不存在，那么它将向数据库中添加这条新的信息。单击“删除”按钮，将从 XML 文档中删除当前显示在窗体中的雇员信息。在雇员列表下拉列表框中选择一个新的雇员，程序将会在下方的框图中添加该雇员的相应信息。单击“清空所有数据”按钮，将会删除雇员列表中的所有雇员信息。

对于将要执行操作的 Employees.xml 文档的具体内容如下所示，其中包含了两条初始成员信息。

```
<?xml version="1.0" standalone="yes"?>
<Employees>
  <Employee EmployeeID="1">
    <Name>小红</Name>
    <Age>25</Age>
    <Salaried>true</Salaried>
    <Wage>800.00</Wage>
    <Active>true</Active>
    <SSN>555-5555</SSN>
    <StartDate>2008-01-01</StartDate>
  </Employee>
  <Employee EmployeeID="2">
    <Name>小明</Name>
    <Age>28</Age>
    <Salaried>true</Salaried>
    <Wage>1000.00</Wage>
    <Active>true</Active>
    <SSN>333-3333</SSN>
    <StartDate>2007-01-01</StartDate>
  </Employee>
</Employees>
```

在文件的起始处，是一个 XML 声明，紧接其后的是一个根元素 <Employees>，每当用户单击“新建”按钮时，将会添加一个新的 <Employee> 元素。这个新添加的 <Employee> 元素将会出现在 <Employees> 元素的子元素的最下面。一个 <Employee> 元素有一个属性

EmployeeID。EmployeeID 属性是由程序自动生成的，在窗体中并不能看见它，而 <Employee> 元素的其他子元素都将与窗体上的内容一一对应。

在本例中，为了统一各元素及属性的名称，在窗体类中先声明以下私有变量：

```
private string TagName = "Name";
private string TagAge = "Age";
private string TagWage = "Wage";
private string TagSSN = "SSN";
private string TagSalaried = "Salaried";
private string TagActive = "Active";
private string TagStartDate = "StartDate";
private string TagEmployees = "Employees";
private string TagEmployee = "Employee";
private string TagEmployeeID = "EmployeeID";
private XmlDocument xmlDocument = null;
```

在以上代码中，最后一条语句声明了一个 XmlDocument 变量，用来表示 XML 文档，这样，在其他方法中对 XML 文档进行处理时可以更方便。

本例中其他窗体相关的代码就不详细列出了，请参考配套光盘中的源码。

21.2.1 创建一个空的 XML 文档

当首次运行本节前面所提到的程序时，将创建一个空的 XML 文档，或者在用户单击“清空所有数据”按钮时，程序也将创建一个空的 XML 文档。

创建一个空的 XML 文档的具体代码如下：

```
private void createEmptyXMLDocument()
{
    //创建一个 XML 文档
    XmlDocument = new XmlDocument();
    //添加 XML 声明
    XmlDeclaration xmlDeclaration = xmlDocument.CreateXmlDeclaration
    ("1.0", "", "yes");
    xmlDocument.PrependChild ( xmlDeclaration );
    //添加根元素
    XmlElement nodeElement = xmlDocument.CreateElement( TagEmployees );
    xmlDocument.AppendChild( nodeElement );
}
```

在上面的程序中，创建了一个 System.Xml.XmlDocument 类的对象，该对象是一个 XML DOM 文档。XmlDocument 类是 System.Xml.XmlNode 类的子类，因此也可以将此文档看成是一个节点。

创建完文档后，则通过 XmlDocument 类的 CreateXmlDeclaration 方法创建了一个 XML 声明节点，并使用 PrependChild 方法将其插入到 XML 文档中的根节点前。

CreateXmlDeclaration 方法共有以下 3 个属性：

- ☐ 版本号，其值必须为 1.0；
- ☐ 编码方式，默认的编码方式是 UTF-8；
- ☐ 指示此文档是否为符合标准的文档，在本例中其值为 yes。

接着，使用 XmlDocument 类的 CreateElement 方法创建本文档的根节点元素，在这个方法中，需要传入根节点元素的标签，在本例中是 Employees。并且通过使用 AppendChild()

方法将这个根节点添加到文档中去。

在执行完上述程序后, XML 文档的内容如下:

```
<?xml version="1.0" standalone="yes"?>
<Employees />
```

添加 XML 声明和根节点元素都是向 XML DOM 文档中添加元素的典型例子, 首先要创建一个想要添加到文档中的节点的实例, 然后将其添加到文档中其他节点的前面或后面。这种添加通常可以通过以下几种方法来实现:

- ☐ AppendChild()方法;
- ☐ InsertAfter()方法;
- ☐ InsertBefore()方法;
- ☐ PrependChild()方法;
- ☐ ReplaceChild()方法。

21.2.2 向 XML 文档添加元素

通过上面的内容, 已经创建了一个空的 XML 文档, 当用户在窗体上添加了所有的雇员信息, 并单击了“保存”按钮后, 则会有一个新的<Employee>元素插入到 XML 文档中。实现这一功能的代码如下:

```
private void addEmployee( XmlDocument doc, int nEmployeeID, string strName,
    string strAge, string strSSN, string strSalaried, string strWage,
    string strActive, string strStartDate )
{
    //生成一个新的<Employee>元素
    //将这个元素添加到根元素的子节点中
    XmlElement nodeParent = doc.DocumentElement;
    XmlElement elemEmployee = doc.CreateElement( TagEmployee );
    elemEmployee.SetAttribute( TagEmployeeID, nEmployeeID.ToString() );
    nodeParent.AppendChild( elemEmployee );
    //为<Employee>元素添加子元素
    addTextElement( doc, elemEmployee, TagName, strName );
    addTextElement( doc, elemEmployee, TagAge, strAge );
    addTextElement( doc, elemEmployee, TagSalaried, strSalaried );
    addTextElement( doc, elemEmployee, TagWage, strWage );
    addTextElement( doc, elemEmployee, TagActive, strActive );
    addTextElement( doc, elemEmployee, TagSSN, strSSN );
    addTextElement( doc, elemEmployee, TagStartDate, strStartDate );
}
private XmlElement addTextElement( XmlDocument doc, XmlElement
nodeParent, string strTag, string strValue )
{
    //通过传入的标签, 创建一个新元素
    XmlElement nodeElem = doc.CreateElement( strTag );
    //通过传入的值, 创建一个文本节点
    XmlText nodeText = doc.CreateTextNode( strValue );
```



```
// 将此元素作为一个子元素添加到一个元素上
nodeParent.AppendChild( nodeElem );
// 将文本节点添加到此元素的子节点中
nodeElem.AppendChild( nodeText );
return nodeElem;
}
```

addEmployee()方法的传入参数有：

- ☐ XML 文档；
- ☐ 一个由程序自动生成的雇员 ID；
- ☐ 在窗体中由用户输入的所有数据。

在 addEmployee()方法中，首先通过 XmlDocument 的 DocumentElement 属性获得了文档的根节点<Employees>元素。然后将所有的<Employee>元素作为子元素添加到根节点<Employees>元素中。

一个空的<Employee>元素通过 CreateElement()方法被创建，而<Employee>元素的 EmployeeID 属性则通过使用 SetAttribute()方法被添加到新的元素中。

最后，使用 AppendChild()方法将<Employee>元素添加到<Employees>元素的子元素后面。

通过调用 addTextElement()方法，可以将窗体中的每一个变量元素添加到<Employee>元素中，这个方法传入的参数是元素标签和与这个元素相关联的字符串值。

在 addTextElement()方法中，首先通过传入的标签项创建了一个空元素，然后通过 CreateTextNode 方法创建了一个文本节点，里面包含了与此元素相关联的窗体上的文本信息。其次，将新的<Employee>元素添加到<Employees>元素的子元素的后面。最后，将包含文本信息的新生成的文本节点通过使用 AppendChild 方法附加到<Employee>元素中。

添加了一个新的雇员信息的 XML 文档如下所示：

```
<?xml version="1.0" standalone="yes"?>
<Employees>
  <Employee EmployeeID="1">
    <Name>小红</Name>
    <Age>25</Age>
    <Salaried>true</Salaried>
    <Wage>800.00</Wage>
    <Active>true</Active>
    <SSN>555-5555</SSN>
    <StartDate>2008-01-01</StartDate>
  </Employee>
</Employees>
```

21.2.3 更新 XML 文档中的元素

在创建了一个新的雇员信息后，可以通过单击“保存”按钮将这个信息保存到文档中。同时，用户也可以通过更改窗体中的雇员信息来更新雇员信息，更改完成后单击“保存”按钮将更新后的信息保存起来。

当用户单击“保存”按钮时，程序将会针对窗体中的新的信息对 XML 文档进行更新。相应的更新源代码如下所示：

```
private void updateEmployee(XmlDocument doc, int nEmployeeID, string
strName, string strAge, string strSSN, string strSalaried, string strWage,
string strActive, string strStartDate )
{
    //查找相应的 employee 元素
    XmlElement empElement = findEmployee( xmlDoc,
nEmployeeID.ToString() );
    if ( empElement == null )
    {
        return;
    }
    //获得 employee 元素的子节点
    XmlNodeList nodeList = empElement.ChildNodes;
    //获得每一个元素的元素标签
    //针对元素标签, 添加文本数据
    for ( int i = 0; i < nodeList.Count; i++ )
    {
        XmlNode node = nodeList.Item( i );
        //程序健壮性检查
        if ( node is System.Xml.XmlElement )
        {
            XmlElement element = (XmlElement) node;
            string strTag = element.Name;
            string strData = "";
            if ( strTag == TagName )
            {
                strData = strName;
            }
            else if ( strTag == TagAge )
            {
                strData = strAge;
            }
            else if ( strTag == TagWage )
            {
                strData = strWage;
            }
            else if ( strTag == TagSSN )
            {
                strData = strSSN;
            }
            else if ( strTag == TagSalaried )
            {
                strData = strSalaried;
            }
            else if ( strTag == TagActive )
            {
                strData = strActive;
            }
            else if ( strTag == TagStartDate )
            {
                strData = strStartDate;
            }
            else
            {
                continue;
            }
        }
    }
}
```



```

        //通过适当的数据创建一个新的文本节点
        //并将其代替现有的文本节点
        XmlText nodeText = doc.CreateTextNode( strData );
        element.ReplaceChild( nodeText, element.FirstChild );
    }
}

private XmlElement findEmployee( XmlDocument doc, string strEmployeeID )
{
    XmlElement nodeFound = null;
    XmlElement root = doc.DocumentElement;
    //获得文档中的所有 employee 元素
    XmlNodeList nodeList = root.GetElementsByTagName( TagEmployee );
    foreach ( XmlNode nodeEmployee in nodeList )
    {
        if ( nodeEmployee is System.Xml.XmlElement )
        {
            //获得 EmployeeID 属性 attribute
            //如果它与要寻找的雇员 ID 相匹配
            //将这个节点保存下来
            XmlElement elemEmployee = (XmlElement) nodeEmployee;
            String strIDFound = elemEmployee.GetAttribute( "EmployeeID" );
            if ( strIDFound != null && strIDFound == strEmployeeID )
            {
                nodeFound = elemEmployee;
                break;
            }
        }
    }
    return nodeFound;
}

```

updateEmployee 方法的传入参数有：

- ☐ XML 文档;
- ☐ 需要更新信息的雇员 ID;
- ☐ 在窗体中由用户输入的所有数据。

在 updateEmployee()方法中，首先通过传入 XML 文档和需要更新信息的雇员 ID 作为参数调用了 findEmployee()方法。

在 findEmployee()方法中，首先通过调用 GetElementsByTagName()方法获得了所有拥有传入的标签值的<Employee>元素的列表。当拥有了所有的<Employee>元素后，则需要通过 findEmployee()方法遍历这个列表，找到由传入参数 Employee ID 所指定的元素。

然后通过下面的代码进行程序的健壮性检查。

```
if ( node is System.Xml.XmlElement )
```

如果得到的节点是一个元素，则可以通过 GetAttribute()方法获得雇员 ID，并将获得的 ID 与传入的 ID 相比较，如果这两个 ID 值相匹配，则程序找到了需要的<Employee>元素，并且返回这个<Employee>元素。否则，循环继续执行，直到找到要找的元素。

回到 updateEmployee()方法，程序检查是否找到了要查找的<Employee>元素。如果找到了，循环这个元素的每个子节点元素，并将由窗体传入的新的数据更新相应的子元素。

首先, 使用<Employee>元素的 ChildNodes 属性获得<Employee>元素的所有的子节点元素, 如下面的代码所示。

```
XmlNodeList nodeList = empElement.ChildNodes;
```

然后, 循环 nodeList 里的每个节点, 并且检查这个节点是否为一个元素。如果是, 则获得这个元素的标签。具体代码如下:

```
for ( int i = 0; i < nodeList.Count; i++ )
{
    XmlNode node = nodeList.Item( i );
    //程序健壮性检查
    if ( node is System.Xml.XmlElement )
    {
        XmlElement element = (XmlElement) node;
        string strTag = element.Name;
```

最后, 基于所获得的元素的标签, 一个被命名为 strData 的字符串变量用来设置获得的标签与给出的标签相吻合的数据。一个包含数据及当前数据所关联新的文本节点将被创建, 并且使用 ReplaceChild()方法代替原有的元素。

```
string strData = "";
if ( strTag == TagFirstName )
{
    strData = strFirstName;
}

// 此处其他检验省略
//...
// 通过适当的数据创建一个新的文本节点
// 并将其代替现有的文本节点
XmlText nodeText = doc.CreateTextNode( strData );
element.ReplaceChild( nodeText, element.FirstChild );
```

21.2.4 删除 XML 文档中的元素

在程序运行窗体中, 单击“删除”按钮, 将从 XML 文档中删除相应的雇员信息。相关的代码如下:

```
private void deleteEmployee( XmlDocument doc, string strEmployeeID )
{
    //在 XML 文档中找到指定的 Employee 元素
    XmlElement element = findEmployee( xmlDoc, strEmployeeID );
    //如果没有找到, 则返回
    if ( element == null )
    {
        return;
    }
    else
    {
        //从 XML 文档中移除元素
        XmlElement root = xmlDoc.DocumentElement;
        root.RemoveChild( element );
    }
}
```


在上面的代码中，首先通过调用 `findEmployee()` 方法来查找相应的 `<Employee>` 元素，如果找到了该元素，那么这个文档的根节点元素将被获得，因为该元素是根节点元素的子元素。`RemoveChild()` 方法可以在 XML 文档中移除该元素。

21.2.5 加载和保存 XML 文档

在本例中，需要使用 `Load()` 方法和 `Save()` 方法来对 XML 文档进行加载和保存。这个功能的代码如下所示。

```
xmlDocument.Load( XMLFileName );
xmlDocument.Save( XMLFileName );
```


不论是 `Load()` 方法还是 `Save()` 方法都有 4 种不同的重载方法，以使 XML 文档的保存和加载具有最大的柔性。

其中，`Load()` 方法可以使用以下变量作为参数。

- ☐ 一个文件流 (Stream)；
- ☐ 文件名称 (string)；
- ☐ `TextReader` 类的实例；
- ☐ `XmlReader` 类的实例。

同样，`Save()` 方法也可以使用相应的 4 个变量作为参数：

- ☐ 一个文件流 (Stream)；
- ☐ 文件名称 (string)；
- ☐ `TextWriter` 类的实例；
- ☐ `XmlWriter` 类的实例。

 说明：这种柔性使 XML 文档可以被不同的数据源创建，包括磁盘文件、缓存和 URL 等。

21.3 用 DataSet 保存 XML 数据

在 .NET 架构中，有一个很重要的类，即 `System.Data.DataSet` 类。这个类主要用来操作保存在存储器里的相关数据。`DataSet` 类是相对独立的，也就是说，`DataSet` 类与数据库之间的关联并不活跃。`DataSet` 里的数据被保存在内存中，在完成对它的操作后获得对数据库的连接，将其中的数据写入到数据库中。因为 `DataSet` 类并不包含与数据库的连接，那么其他数据源也可以使用并加载 `DataSet`。XML 就是一个这样的数据源。本节将讲述 XML 文档与 `DataSet` 类之间的关系。

在本节中，示例程序需要以下几个文件：

- ☐ `employee1.xml`；
- ☐ `wagehistory.xml`；
- ☐ `employee.xsd`；
- ☐ `wagehistory.xsd`。

`employee1.xml` 的文件内容如下所示。

```
<?xml version="1.0" standalone="yes"?>
<Employees>
  <Employee EmployeeID="1">
    <Name>小红</Name>
    <Age>25</Age>
    <Salaried>true</Salaried>
    <Wage>800.00</Wage>
    <Active>true</Active>
    <SSN>555-5555</SSN>
    <StartDate>2008-01-01</StartDate>
  </Employee>
  <Employee EmployeeID="2">
    <Name>小明</Name>
    <Age>28</Age>
    <Salaried>true</Salaried>
    <Wage>1000.00</Wage>
    <Active>true</Active>
    <SSN>333-3333</SSN>
    <StartDate>2007-01-01</StartDate>
  </Employee>
</Employees>
```

wagehistory.xml 的文件内容如下所示。

```
<?xml version="1.0" standalone="yes"?>
<WageHistory>
  <WageChange EmployeeID="1">
    <Date>2008-01-01</Date>
    <Wage>800.00</Wage>
  </WageChange>
  <WageChange EmployeeID="2">
    <Date>2007-01-01</Date>
    <Wage>1000.00</Wage>
  </WageChange>
</WageHistory>
```

通过比较以上两个文件可以发现，在这两个文件中，无论是<Employee>元素还是<WageHistory>元素都有 EmployeeID 属性，也就是说 EmployeeID 属性确定了两个 XML 文档间的关联。在 employee1.xml 中每一个<Employee>元素在 wagehistory.xml 中有一个或更多个<WageChange>元素。这个关联将被应用在下面的示例程序中，来说明每一个职员工资增长信息。

下面的示例文档将给出 3 种独立的操作。首先，它将不使用 XML schema 文件加载两个 XML 文档，并建立起两个文件间的关联。然后，它将使用 XML schema 文件加载两个 XML 文档，并建立起两个文件间的关联。最后，它将使用 XML schema 文件加载两个 XML 文档，并以编程的方式获得建立了关联的文档。

如图 21.2 所示为用户单击“不使用 Schema 文件加载 XML 文档”按钮后应用程序窗体的外观。程序将 wagehistory.xml 文件和 employee1.xml 文件中的数据读入到一个 DataSet 类的对象中，并填充到“雇员信息表”和“工资变动表”中。

“雇员信息表”就是位于应用程序窗体中的上方的表格，而“工资变动表”则是位于应用程序窗体中间位置的表格。在前面的学习中可以知道，表格可以自动根据 DataSet 中的数据进行填充，并且生成它们之间的联系。用户可以单击“雇员信息表”或“工资变动表”中的任意一条信息，而另一个表与之相关联的数据行将会突出显示，并不需要任何附

加的程序。

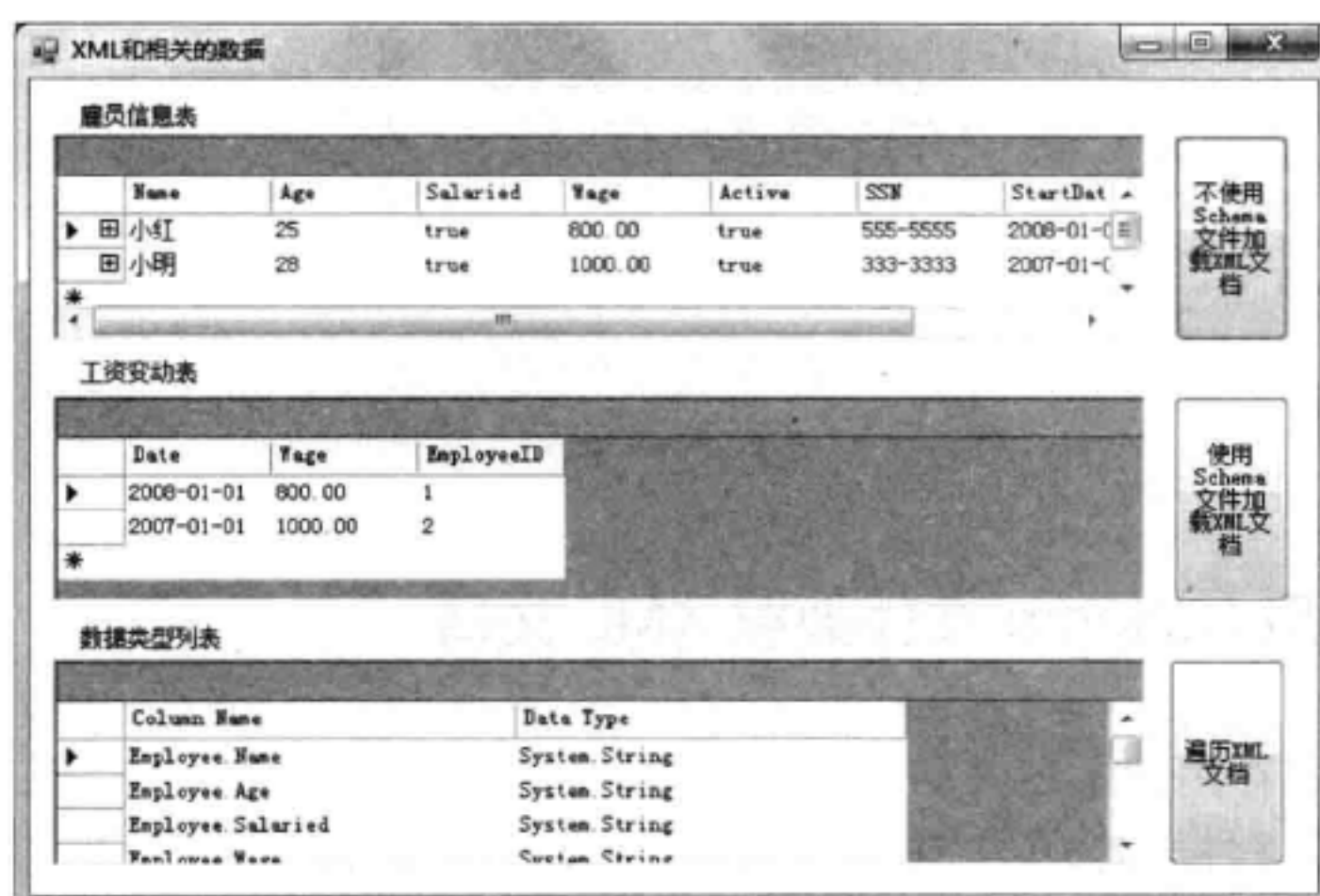


图 21.2 不使用 Schema 文件加载 XML 文档

应用程序底部的表格则显示了在“雇员信息表”以及“工资变动表”中的每一列数据的数据类型，在图 21.2 中可以看出，每一个数据类型都是字符串（string）的类型。而在单击了“使用 Schema 文件加载 XML 文档”按钮后，应用程序的运行界面如图 21.3 所示，可以看到在应用程序窗体的最下面的表格中，一些列的数据类型发生了变化，这正是使用 XSD Schema 将 XML 中的数据读入 DataSet 对象时所产生的影响。Schema 文件确定了 XML 文档中每一个元素和属性的数据类型。这些数据类型可以被 DataSet 类传递，比较图 21.2 和图 21.3 可以发现，在应用程序顶部的表格中，Salaried 和 Active 列由文本框变成了复选框。

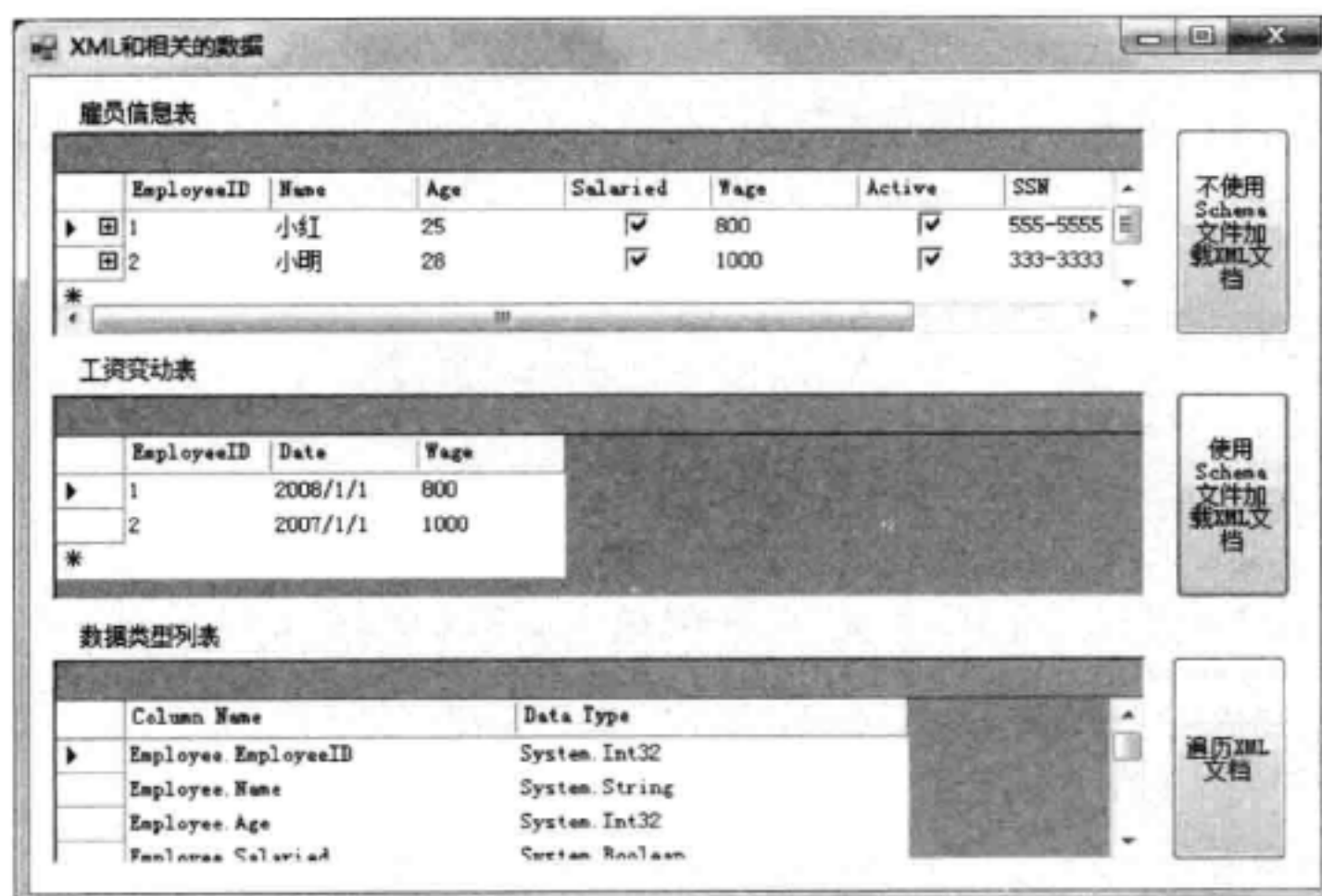


图 21.3 使用 Schema 文件加载 XML 文档

如图 21.4 所示为当用户单击“遍历 XML 文档”按钮时，对 DataSet 中的数据进行遍历的结果。

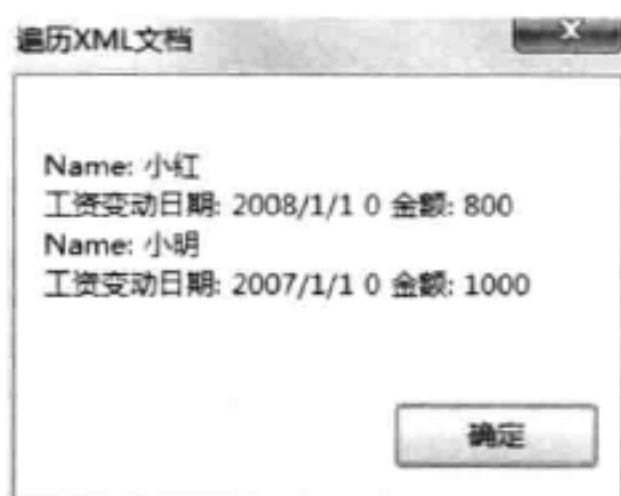


图 21.4 “遍历 XML 文档” 文本框

21.3.1 不使用 Schema 文件加载 XML 文档

下面的代码将在用户单击“不使用 Schema 文件加载 XML 文档”按钮时，加载 XML 文档并建立起两个文件间的关联。图 21.2 是单击按钮后所显示的运行结果，其执行代码如下：

```

/// <summary>
/// 当“不使用文件加载 XML 文档”按钮被单击时被调用
/// </summary>
private void button1_Click(object sender, System.EventArgs e)
{
    Cursor currentCursor = Cursor.Current;
    try
    {
        Cursor.Current = Cursors.WaitCursor;
        //生成两个新的 dataset
        //从 XML 文档中向这两个 dataset 加载数据
        DataSet dsEmployees = new DataSet( "Employees" );
        DataSet dsWageHistory = new DataSet( "WageHistory" );
        loadAndDisplayDatasets( dsEmployees, dsWageHistory );
    }
    catch ( Exception exception )
    {
        MessageBox.Show( "出现异常" );
    }
    finally
    {
        Cursor.Current = currentCursor;
    }
}
/// <summary>
/// 从 XML 文档中向 dataset 加载数据，将数据显示在窗体中
/// </summary>
/// <param name="dsEmployees"> employees 的 dataset</param>
/// <param name="dsWageHistory"> wagehistory 的 dataset</param>
private void loadAndDisplayDatasets( DataSet dsEmployees, DataSet
dsWageHistory )
{
    //从 XML 文档中向 dataset 加载数据
    dsEmployees.ReadXml( "employee1.xml" );
    dsWageHistory.ReadXml( "wagehistory.xml" );
    //将 WageChange 表格复制至 Employees 的 dataset
    dsEmployees.Tables.Add(dsWageHistory.Tables["WageChange"].Copy() );
    DataTable tblEmp = dsEmployees.Tables["Employee"];
    DataTable tblWageHistory = dsEmployees.Tables["WageChange"];
}

```



```

//基于 employee ID 创建两个表间的关联
DataRelation relation = new DataRelation("EmpWageHistory",
                                         new DataColumn[] {tblEmp.Columns
                                         ["EmployeeID"]},
                                         new DataColumn[] {tblWageHistory.
                                         Columns["EmployeeID"]},
                                         false);
dsEmployees.Relations.Add( relation );
//将 dataset 绑定到 DataGrid
gridEmployees.SetDataBinding( dsEmployees, "Employee" );
gridRaises.SetDataBinding( dsEmployees, "WageChange" );

DataTable tblDataTypes = new DataTable("DataTypes");
DataColumn col1 = new DataColumn("Column Name");
DataColumn col2 = new DataColumn("Data Type");
tblDataTypes.Columns.Add(col1);
tblDataTypes.Columns.Add(col2);
DataSet dsDataTypes = new DataSet();
dsDataTypes.Tables.Add(tblDataTypes);

//遍历 dsEmployees 里的表
foreach ( DataTable table in dsEmployees.Tables )
{
    string strTableName = table.TableName;
    //遍历每个表中的列
    foreach ( DataColumn column in table.Columns )
    {
        //读取每个列的内容
        string strName = strTableName + "." + column.ColumnName;
        string strDataType = column.DataType.ToString();
        //向表中添加数据
        DataRow row = tblDataTypes.NewRow();
        row["Column Name"] = strName;
        row["Data Type"] = strDataType;
        tblDataTypes.Rows.Add( row );
    }
}
//设置表的外观
gridDataTypes.PreferredColumnWidth = 200;
gridDataTypes.SetDataBinding( dsDataTypes, "DataTypes" );
}

```

在用户单击了“不使用 Schema 文件加载 XML 文档”按钮后，将生成两个空的 DataSet 对象，一个用来存储雇员信息，另一个用来存储工资历史信息。然后程序将调用 loadAndDisplayDatasets() 方法，并传入新近生成的 DataSet 对象。

loadAndDisplayDatasets() 方法才是真正完成业务逻辑的函数。首先，通过调用 DataSet 对象的 ReadXml() 方法来向 DataSet 加载 XML 文档中的内容。然后，通过以下代码，将工资历史信息从工资历史信息的 DataSet 复制到雇员信息的 DataSet。

```

//将 WageChange 表格复制至 Employees 的 dataset
dsEmployees.Tables.Add(dsWageHistory.Tables["WageChange"].Copy() );

```

这一步非常重要，因为 DataSet 类的 ReadXml() 方法在加载 XML 文件时只能被调用一次。因此，程序首先将每一个文件加载到一个单独的 DataSet 对象中，然后再将工资历史信息复制到雇员 DataSet 对象中。最后，程序将建立雇员信息和工资数据信息之间的关联，可以使用如下代码实现这一功能：


```

DataTable tblEmp = dsEmployees.Tables["Employee"];
DataTable tblWageHistory = dsEmployees.Tables["WageChange"];
//基于 employee ID 创建两个表间的关联
DataRelation relation = new DataRelation("EmpWageHistory",
    new DataColumn[] {tblEmp.Columns
        ["EmployeeID"]}, new DataColumn[]
        {tblWageHistory.Columns ["EmployeeID"]},
        false);
dsEmployees.Relations.Add( relation );


```

首先,在 DataSet 对象 dsEmployees 中将创建关于雇员信息表和工资变动表的引用,并将其传入 DataRelation 的构造函数中。通过实例化 System.Data.DataRelation 类来创建一个新的关系。构造函数的第 1 个参数是新关系的名字,在本例中是 EmpWageHistory;第 2 个参数指出了构成父关系的表中的列,本例中是雇员信息表中的 EmployeeID;第 3 个参数指出了构成子关系表中的列,本例中是工资变动表中的 EmployeeID;最后一个参数指示了这个关系是否有某些限制,比如唯一性或者外键,在本例中,数据是只读的,所以不需要添加任何限制。

在新的关系被创建后,需要将它添加到 DataSet 对象中,可以通过调用 dsEmployees 对象的 Relations.Add()方法来完成这个功能。最后,将 dsEmployees 的雇员信息表添加到窗体顶部的表格中,使雇员信息显示在应用程序窗体顶部的表格中。将新创建的关系添加到窗体中间的表格中,使工资信息在此表格中显示出来。

剩下的代码创建了一个新的 DataSet 对象,在这个对象中包含了每一列的列名和该列的数据类型,然后将这个新生成的 DataSet 对象绑定到应用程序窗体下方的表格控件中。则这个表格将显示所有表格的各个列的列名及其相应的数据类型列表。

从图 21.2 中可以发现,对于窗体中的每一列所对应的数据类型都是 System.String。之所以会产生这种结果,是因为 DataSet 对象在程序没有指出 Schema 文件时,将会自己创建一个 Schema 文件。创建的 Schema 文件规定了每一个元素和属性在 XML 文档中的数据类型。但是程序不能正确判断出每一个元素和属性的类型,因为在 XML 文档中,任何元素和属性都是文本结构的,所以在程序自己创建的 Schema 文件中,所有的元素和属性都是 System.String 类型。

 **注意:** 如果需要得到每一个元素和属性的正确的数据类型,则需要指示给 DataSet 对象,导入 XML 文档数据时使用的 XSD 文件。

21.3.2 使用 Schema 文件加载 XML 文档

在向 DataSet 对象载入数据时,可以通过指定相应的 XML XSD 文件来指出每一个元素和属性所对应的数据类型。并且当指定了 XML XSD 文件时,表格中的列会根据数据类型自动生成相对应的列类型。在下面的代码中,给出了使用模式文件加载数据时程序的运行逻辑(也就是用户单击“使用 Schema 文件加载 XML 文档”按钮后所运行的程序)。

```

/// <summary>
/// 用户单击“使用 Schema 文件加载 XML 文档”按钮后所运行的程序
/// </summary>
private void button2_Click(object sender, System.EventArgs e)
{

```



```

Cursor currentCursor = Cursor.Current;
try
{
    Cursor.Current = Cursors.WaitCursor;
    //生成两个新的 datasets
    //加载 XML 文档中的数据
    DataSet dsEmployees = new DataSet( "Employees" );
    DataSet dsWageHistory = new DataSet( "WageHistory" );
    dsEmployees.ReadXmlSchema( "employees.xsd" );
    dsWageHistory.ReadXmlSchema( "wagehistory.xsd" );
    loadAndDisplayDatasets( dsEmployees, dsWageHistory );
}
//捕获异常
catch ( Exception exception )
{
    *
    MessageBox.Show("出现异常");
}
//恢复光标形态
finally
{
    Cursor.Current = currentCursor;
}
}

```

程序的运行结果如图 21.3 所示。

比较这段代码与上一段代码的区别是，上述代码使用了两遍 DataSet 类的 ReadXmlSchema()方法来读取 Schema 文件。Employees.xsd 文件的内容如下所示：

```

<xsd:schema id="Employees" targetNamespace="" xmlns="" xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xsd:element name="Employees" msdata:IsDataSet="true">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="Employee">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Name" type="xsd:string" minOccurs="0" />
              <xsd:element name="Age" type="xsd:int" minOccurs="0" />
              <xsd:element name="Salaried" type="xsd:boolean" minOccurs="0" />
              <xsd:element name="Wage" type="xsd:double" minOccurs="0" />
              <xsd:element name="Active" type="xsd:boolean" minOccurs="0" />
              <xsd:element name="SSN" type="xsd:string" minOccurs="0" />
              <xsd:element name="StartDate" type="xsd:date" minOccurs="0" />
            </xsd:sequence>
            <xsd:attribute name="EmployeeID" type="xsd:int" />
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

WageHistory.xsd 文件的内容如下所示：

```

<?xml version="1.0" standalone="yes"?><xsd:schema id="Employees"
targetNamespace="" xmlns="" xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

```

```
<xsd:element name="Employees" msdata:IsDataSet="true">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="WageChange">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Date" type="xsd:date" minOccurs="0" />
            <xsd:element name="Wage" type="xsd:double" minOccurs="0" />
          </xsd:sequence>
            <xsd:attribute name="EmployeeID" type="xsd:int" />
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

比较图 21.2 和图 21.3 的输出结果可以看到，在应用窗体底部的表格中，wage 元素的类型变成了 double 型，而在 Employees.xsd 文件中，wage 元素的类型也被下列语句标注为 double 型。

```
<xsd:element name="Wage" type="xsd:double" minOccurs="0" />
```

21.3.3 遍历 XML 文档

由于网格控件是一个数据绑定控件，则它能够理解在 DataSet 对象中的关系，并且能够在这些关系的基础上显示这些数据。因此，在前面的例子中，并不需要专门的编写代码来建立两个 XML 文档中数据的关系，当用户单击“遍历 XML 文档”按钮时，所运行的代码将在下面显示。

在 .NET Framework 中，提供了一个名为 System.Xml.XmlDataDocument 的类，这个 XmlDataDocument 类允许通过使用 XML DOM 方法访问 XML 文档中的数据。XmlDataDocument 类有一个 DataSet 类型的成员变量，使开发人员可以调用 DataSet 成员。由于 DataDocument 类继承自 XmlDocument 类，所以同样也可以使用 XML DOM 方法调用 DataSet 成员。

在下面的示例程序中，给出了使用以上两种方法访问文档的具体程序。首先来看一下在代码中加载 XmlDataDocument 实例的方法：

```
/// <summary>
/// 加载两个 XML 文档中的数据进入 XmlDataDocument 类
/// 建立两个文档间的关系
/// </summary>
/// <returns>XmlDataDocument</returns>
private XmlDataDocument loadXML()
{
    // 加载雇员信息 XML 文档
    DataSet dsEmployees = new DataSet( "Employees" );
    dsEmployees.ReadXmlSchema( "employees.xsd" );
    dsEmployees.ReadXml( "employee1.xml" );
    dsEmployees.Tables[0].TableName = "Employee";
    // 加载工资信息 XML 文档
    DataSet dsWageHistory = new DataSet( "WageHistory" );
    dsWageHistory.ReadXmlSchema( "wagehistory.xsd" );
    dsWageHistory.ReadXml( "wagehistory.xml" );
}
```



```

dsWageHistory.Tables[0].TableName = "WageChange";
//将工资信息复制到雇员 dataset
dsEmployees.Tables.Add(dsWageHistory.Tables["WageChange"].Copy() );
DataTable tblEmp = dsEmployees.Tables["Employee"];
DataTable tblWageHistory = dsEmployees.Tables["WageChange"];
//基于 employee ID 创建两个表间的关联
DataRelation relation = new DataRelation("EmpWageHistory",
                                         new DataColumn[] {tblEmp.Columns
                                                             ["EmployeeID"]}, new DataColumn[]
                                         {tblWageHistory.
                                         Columns["EmployeeID"]}, false);

//设置 relation 的 Nested 属性为 true
relation.Nested = true;
//添加 relation
dsEmployees.Relations.Add( relation );
//实例化 XmlDataDocument
XmlDataDocument doc = new XmlDataDocument( dsEmployees );
return doc;
}

```

该段代码与前面的一段代码很相似。在开始处，将两个 XML 文档中的数据加载到两个独立的 DataSet 对象中，然后将工资信息复制到雇员信息 DataSet 中，则两个文件数据间的关系将被创建并加载到 DataSet 中。最后，将会有有一个 XmlDataDocument 对象被创建。

在上面的代码中，将 DataRelation 实例的 Nested 属性设置为 true，这样可以使程序自动创建两个文档间的关系，并且工资变动信息表中的信息将是其相对应的雇员信息的子信息。如果 Nested 属性设置为 false，那么工资变动表中的元素将紧接着雇员信息的元素下方添加并显示。

下面这段代码是使用相关方法调用来遍历整个文档。

```

/// <summary>
///利用相关调用遍历 XML 文档
/// 将职员的信息展示在显示屏上
/// </summary>
/// <param name="doc"> XML 文档</param>
private void retrieveAsData( XmlDataDocument doc )
{
    //自定义变量
    string strName;
    string strAge;
    string strLastName;
    string strDate;
    string strWage;
    //使用查找方法来获得相关数据
    DataTable tblEmp = doc.DataSet.Tables["Employee"];
    DataRelation relation = doc.DataSet.Relations["EmpWageHistory"];
    DataRow[] rows = tblEmp.Select();
    for( int i = 0; i < rows.Length; i ++ )
    {
        //获得雇员信息
        DataRow rowEmp = rows[i];
        strName = rowEmp[1].ToString();
        strOutput += "Name: " + strName + "\r\n";
        DataRow[] rowsWage = rowEmp.GetChildRows( relation );
        //遍历工资信息
        for( int j = 0; j < rowsWage.Length; j++ )
        {

```

```

        DataRow rowWage = rowsWage[j];
        strDate = rowWage[1].ToString();
        strWage = rowWage[2].ToString();
        strOutput += "工资变动日期: " + strDate.Substring(0, 10);
        strOutput += " 金额: " + strWage + "\r\n";
    }
}
}

```

在上面的这段代码中，首先调用 `Select()` 方法来获得 `DataTable` 里的数据，具体代码如下：

```

DataTable tblEmp = doc.DataSet.Tables["Employee"];
DataRow[] rows = tblEmp.Select();

```

在这个例子里，将会返回一个 `DataRow` 类型的实例，里面包含了 `DataTable` 所有的数据信息。可以使用以下代码获得 `EmployeeID` 为 1 的雇员的名字：

```

DataRow rowEmp = rows[i];
strName = rowEmp[1].ToString();

```

而 `relation` 对象的信息则在程序的前部使用如下代码进行设置：

```

DataRelation relation = doc.DataSet.Relations["EmpWageHistory"];

```

21.4 用 XPath 查找节点

XPath (XML path Language, XML 路径语言)，它是一种用来确定 XML 文档中某部分位置的语言，XPath 基于 XML 的树状结构，提供在数据结构树中找寻节点的能力。

21.4.1 XPath 简介

在本章的 21.1 节中，介绍了 XPath。XPath 是用来在一定的范围内按照一个匹配规则在 XML 文档中查找节点。

在 XML DOM 中，假如想要找到一个指定的元素，是一件非常枯燥、繁琐的事情，程序必须在内存里遍历整棵 DOM 树，直到遇到要查找的节点。在 XML DOM 中，不能够根据给出的一组字符串直接找到相应的元素，但是在 XPath 语言中，却能够提供这种功能。

21.4.2 XPath 查询示例代码

在下例中，将会演示如何使用 XPath 语句来查询 XML 文档。在这个演示程序中，将使用 `person1.xml` 文件。该文件的具体内容如下：

```

<?xml version="1.0" standalone="yes"?>
<Employees>
  <Employee EmployeeID="1">
    <Name>小佳</Name>
    <Age>28</Age>
    <Salaried>true</Salaried>
    <Wage>40000</Wage>
  </Employee>

```



```

<Active>true</Active>
<Title>程序员</Title>
<Location>
  <Address>42 ZhongShan Road</Address>
  <City>HaerBin</City>
  <Province>HeiLongJiang</Province>
  <Zip>150001</Zip>
</Location>
</Employee>
<Employee EmployeeID="2">
  <Name>小强</Name>
  <Age>28</Age>
  <Salaried>>false</Salaried>
  <Wage>30000</Wage>
  <Active>true</Active>
  <Title>程序员</Title>
  <Location>
    <Address>97 YanHe Avenue</Address>
    <City>HaerBin</City>
    <Province>HeiLongJiang</Province>
    <Zip>150001</Zip>
  </Location>
</Employee>
</Employees>

```

首先创建一个 Windows 窗体，在其中添加一个文本框控件和一个按钮控件。在文本框控件中将显示程序的查询结果。将按钮的文本设置为“运行 XPath 查询”。应用程序窗体外观如图 21.5 所示。



图 21.5 应用程序窗体外观

在用户单击“运行 XPath 查询”按钮后，程序将进行一系列的针对 personel.xml 文件的查询，并将结果写入窗体中的文本框控件中。

下面的代码演示了如何对 personel.xml 文件生成查询。

```

//引用的命名空间
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Xml;

```

```

using System.Xml.XPath;
using System.IO;
namespace XPath
{
    public class Form1 : System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components = null;
        private System.Windows.Forms.TextBox textBox1;
        private System.Windows.Forms.Button button1;
        private string strOutput;
        public Form1()
        {
            InitializeComponent();
        }
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
        /// <summary>
        /// 在需要 XPath 查询时调用
        /// </summary>
        private void button1_Click(object sender, System.EventArgs e)
        {
            Cursor currentCursor = Cursor.Current;
            try
            {
                Cursor.Current = Cursors.WaitCursor;
                // 在 XPath 文档和 XML DOM 文档中进行查询
                doXPathDocumentQueries();
                doXmlDocumentQueries();
                // 在屏幕上显示结果
                textBox1.Text = strOutput;
            }
            // 捕获异常
            catch (Exception exception)
            {
                MessageBox.Show(exception.Message);
            }
            finally
            {
                Cursor.Current = currentCursor;
            }
            /// <summary>
            /// 对只读的 XPathDocument 文档进行 XPath 查询
            /// </summary>
            private void doXPathDocumentQueries()
            {
                strOutput = "**** 开始对 XPathDocument 文档进行查询***\r\n\r\n";
                // 将 XML 文档加载进一个只读的 XPathDocument
                // 实例化 XPathNavigator
                XPathDocument doc = new XPathDocument("person1.xml");
                XPathNavigator navigator = doc.CreateNavigator();
                strOutput += "**** 显示所有工资信息 ***\r\n\r\n";
                // 在文档中查找到雇员工资元素
                // 将工资信息显示在显示屏上
                XPathNodeIterator iterator = navigator.Select(
                    "descendant::Employee/Wage");
                while (iterator.MoveNext())
            }
        }
    }
}

```



```

{
    strOutput += iterator.Current.Name + ": ";
    strOutput += iterator.Current.Value + "\r\n";
}
strOutput += "\r\n\r\n*** 显示在黑龙江的所有雇员 ***\r\n\r\n";
//找到所有在黑龙江办公室的雇员
//将这些雇员的名字显示在显示屏上
iterator = navigator.Select( "//Employee[Location/Zip='150001']" );
while ( iterator.MoveNext() )
{
    XPathNavigator nav2 = iterator.Current;
    nav2.MoveToFirstChild();
    //获得雇员名称
    strOutput += nav2.Value;
}
strOutput += "\r\n\r\n*** 有薪雇员的平均工资 ***\r\n\r\n";
//计算公司里所有有薪雇员的平均工资
//将计算结果显示在显示屏上
Int32 nAverage = (Int32)(Double)navigator.Evaluate(
    "sum(//Employee[Salaried='true']/Wage) div
    count(//Employee[Salaried='true'])" );
strOutput += "平均工资: ¥" + nAverage.ToString();
}
/// <summary>
///在 XML DOM 文档上进行 XPath 查询
///修改相应文档
/// </summary>
private void doXmlDocumentQueries()
{
    strOutput += "\r\n\r\n*** 开始 XML 文档查询***\r\n\r\n";
    //向一个 DOM XML 文档中加载 XML 文档
    XmlDocument doc = new XmlDocument();
    doc.Load( "personel.xml" );
    //对于每一个黑龙江雇员获得一系列的活动节点
    XmlNodeList nodeList = doc.SelectNodes(
        "//Employee[Location/Zip='97206']/Active");
    foreach ( XmlNode node in nodeList )
    {
        //将每一个黑龙江的雇员设置为不活动的
        node.InnerText = "false";
    }
    //将修改后的文档显示在显示屏上
    StringWriter writerString = new StringWriter();
    XmlTextWriter writer = new XmlTextWriter( writerString );
    writer.Formatting = Formatting.Indented;
    doc.WriteTo( writer );
    writer.Flush();
    strOutput += writerString.ToString();
}
}
}

```

程序将运行两组基于文件的 XPath 的查询。第一组查询是将 XML 文档加载到一个 System.XML.XPath.XPathDocument 的对象中去。XPathDocument 类将会优化 XPath 查询, 它支持只读的访问文档。XPathNavigator 类的实例化对象用来完成对于 XPathDocument 类的查询任务。

21.4.3 XPath 示例代码讲解


在 21.4.2 节中介绍的程序示例中,通过下面的代码实例化一个 `XPathDocument` 类的对象,并将磁盘中的 XML 文件加载到该对象中,该对象用来执行对文档的查询。

```
XPathDocument doc = new XPathDocument( "person1.xml" );
XPathNavigator navigator = doc.CreateNavigator();
```

`XPathNavigator` 类的查找方法用来执行一个对 XML 文档的查询,它使用一个 XPath 语句作为一个参数,下面的程序将返回在 XML 文档中的所有<Wage>元素。

```
XPathNodeIterator iterator = navigator.Select(
    "descendant::Employee/Wage" );
//遍历
while ( iterator.MoveNext() )
{
    strOutput += iterator.Current.Name + ": ";
    strOutput += iterator.Current.Value + "\r\n";
}
```

`Select` 语句使用 XPath 表达式作为一个参数,并且返回一个 `XPathNodeIterator` 类的对象,该对象是一个包含了所有节点的列表。`XPathNodeIterator` 的 `Current` 属性指出了 `Select()` 方法返回的节点列表中的当前位置。在调用 `MoveNext()` 方法前,此位置是没有定义的。对于每一个<Wage>元素都会存在于这个返回的列表中,而元素的标签和值都将被保存,以便在后面的程序中显示在显示屏上。

 **说明:** `Current` 属性将返回一个元素标签,作为此标签所对应的值。

对于 `XPathDocument` 对象的所做的第二个查询将返回所有在黑龙江办公的雇员信息,并在显示屏上显示这些雇员的名字。下面的程序段将完成以下功能:

```
iterator = navigator.Select( "//Employee[Location/Zip='150001']" );
while ( iterator.MoveNext() )
{
    XPathNavigator nav2 = iterator.Current;
    nav2.MoveToFirstChild();
    //获得雇员名称
    strOutput += nav2.Value;
}
```

这个查询的区别仅在于,开发人员需要使用另一个 `XPathNavigator` 类的实例。在 `node` 列表中的每一个节点都是一个<Employee>元素。第二个 `XPathNavigator` 对象是必须的,它用来指出当前程序在<Employee>元素列表中的位置。

对于 `XPathDocument` 对象的最后一个查询是一个摘要查询。它返回的是一个结果,而不是一个列表。例如下面的代码,返回的就是所有有酬员工的平均工资。

```
Int32 nAverage = (Int32)(Double)navigator.Evaluate(
    "sum(//Employee[Salaried='true']/Wage) div
    count(//Employee[Salaried='true'])" );
```

当进行摘要查询时, `Evaluate()` 方法用来代替 `Select()` 方法。`Evaluate()` 方法同样使用

XPath 表达式作为一个参数，在上面的例子中可以看出，给出的做为参数的表达式可以非常复杂。

第二组查询主要是针对 XML 文本对象。正如前面提到的，XPathDocument 对象是一个只读类型的对象。所以，如果想要对一个 XML 文档直接使用 XPath 语言并更新文档，则需要一个 XmlDocument 对象。下面是这个例子中的相关代码：

```
XmlDocument doc = new XmlDocument();
doc.Load( "personnel.xml" );
//对于每一个黑龙江雇员获得一系列的活动节点
XmlNodeList nodeList = doc.SelectNodes("//Employee[Location/
Zip='97206']/Active");
foreach ( XmlNode node in nodeList )
{
    //将每一个黑龙江的雇员设置为不活动的
    node.InnerText = "false";
}
```

上面的代码模仿关闭了黑龙江的办公室，将所有办公地址在黑龙江的雇员的<Active>元素置为 false。首先，实例化一个 XmlDocument 对象，然后使用 Load()方法加载。其次，使用 System.Xml.Node 类的 SelectNodes()方法对于文档进行 XPath 查询。它传入一个 XPath 表达式作为参数，并且返回一个 XmlNodeList 对象。在这个例子中，这个 XmlNodeList 对象是一个节点列表，包含每一个在黑龙江工作的雇员的<Active>元素。可以使用 foreach 语句对这个列表进行遍历，并将<Active>元素的文本设为 false。

21.5 使用 XSL 将 XML 文档转化为另一种格式

XSLT 同样利用了 XPath 表达式。XSLT 可以用来将 XML 文档从一种格式转化为另一种格式。结果格式可以是另一个 XML 文档、HTML 或其他文本格式。在转换时，XSLT 样式表通常使用 XPath 语句在 XML 文档中定位节点。

在 21.4.3 节中，演示了如何将 XPath 作为一个查询工具使用。本节中，将主要讲解如何使用 XSLT 来将 XML 文档转化为另一种格式。XSL 样式表使用 XPath 表达式从选取节点链表进行转换。

21.5.1 XSL 转换示例一

在下面的例子中，将开发一个名为“曙光”公司的人力资源管理分系统，该公司的人事信息将记录在一个名为 personnel.xml 的文件中。该公司的另一个分系统维持着一个网站，这个网站包括一些基于人力资源的报表。网站使用 XSL 格式表将包含人员信息的 XML 文档转化为 HTML 网页文件。在下面的例子中，网站的格式表期望 XML 文档使用另外一种

不同于 personel.xml 文件的格式,所以在示例代码中,需要先将 personel.xml 文件转化为网站格式表所希望输入的格式,再使用 reports.XSL 样式表来将其转化为常见的 HTML 格式。

在本例中需要的文件有:

- ☐ personel.xml;
- ☐ salariedpersonel.xsl;
- ☐ salariedreport.xsl。

下面分别对这 3 个文件的内容进行介绍。


personel.xml 文件内容如下所示。

```
<?xml version="1.0" standalone="yes"?>
<Employees>
  <Employee EmployeeID="1">
    <FirstName>Jane</FirstName>
    <LastName>Lee</LastName>
    <Age>28</Age>
    <Salaried>true</Salaried>
    <Wage>40000</Wage>
    <Active>true</Active>
    <Title>Programmer</Title>
    <Location>
      <Address>42 ZhongShan Road</Address>
      <City>HaerBin</City>
      <Province>HeiLongJiang</Province>
      <Zip>150001</Zip>
    </Location>
  </Employee>
  <Employee EmployeeID="2">
    <FirstName>Force</FirstName>
    <LastName>Fu</LastName>
    <Age>28</Age>
    <Salaried>true</Salaried>
    <Wage>30000</Wage>
    <Active>true</Active>
    <Title>Programmer</Title>
    <Location>
      <Address>97 YanHe Avenue</Address>
      <City>HaerBin</City>
      <Province>HeiLongJiang</Province>
      <Zip>150001</Zip>
    </Location>
  </Employee>
  <Employee EmployeeID="3">
    <FirstName>Trancy</FirstName>
    <LastName>Ding</LastName>
    <Age>28</Age>
    <Salaried>true</Salaried>
    <Wage>20000</Wage>
    <Active>true</Active>
    <Title>Programmer</Title>
    <Location>
      <Address>18 ZhiChunRoad</Address>
      <City>BeiJing</City>
      <Province>BeiJing</Province>
      <Zip>100083</Zip>
    </Location>
  </Employee>
</Employees>
```


转换得到的 `salaried.xml` 文件内容如下所示。

```
<Salaried>
  <Employee>
    <Name>Jane Lee</Name>
    <Wage>40000</Wage>
    <Location>150001</Location>
  </Employee>
  <Employee>
    <Name>Force Fu</Name>
    <Wage>30000</Wage>
    <Location>150001</Location>
  </Employee>
</Salaried>
```

如上所示，XSL 样式表将 `<FirstName>` 与 `<LastName>` 整合到一起，形成一个完整的 `<Name>` 元素。同样，XSL 样式表也将复制 `<Wage>` 元素。最后，它需要复制 `<Zip>` 元素的内容到 `<Location>` 元素。

 说明： `salariedpersonel.xml` 文件是将第一个 XML 文件转化成第二个 XML 文件所使用的样式表。 `salariedpersonel.xml` 文件内容如下所示。


```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <Salaried>
      <xsl:apply-templates/>
    </Salaried>
  </xsl:template>
  <xsl:template match="Employees">
    <xsl:apply-templates select="Employee[Salaried='true']"/>
  </xsl:template>
  <xsl:template match="Employee[Salaried='true']">
    <Employee>
      <Name>
        <xsl:value-of select="FirstName"/><xsl:text> </xsl:text>
        <xsl:value-of select="LastName"/>
      </Name>
      <Wage>
        <xsl:value-of select="Wage"/>
      </Wage>
      <Location>
        <xsl:value-of select="Location/Zip"/>
      </Location>
    </Employee>
  </xsl:template>
</xsl:stylesheet>
```

一个 XSL 样式表使用模式匹配来处理 XML 文档。XSLT 处理器开始处理 XML 文档，并且在 XSL 脚本中查找语句，调用规则，配比在 XML 文档中遇到的节点。下面的语段是生成文档的根元素的规则：

```
<xsl:template match="/">
  <Salaried>
    <xsl:apply-templates/>
  </Salaried>
</xsl:template>
```

当遇到原始的 XML 文档的根节点时, 之前的语句将被执行。在这个例子中, <Salaried> 元素被创建, 并且根元素的子节点被 <xsl:apply-templates/> 语句处理。当遇到 <Employees> 元素时, 将执行如下所示的语句。

```
<xsl:template match="Employees">
  <xsl:apply-templates select="Employee[Salaried='true']"/>
</xsl:template>
```

 说明: 查找属性的 <xsl:apply-templates/> 语句包含的模式应与 XML 文档相匹配。模式匹配字符串应该是一个 XPath 表达式。这也正是 XPath 在 XSL 样式表中所起的作用。

前面的 XSL 脚本语句忽略了 <Salaried> 子元素的值为 true 的 <Employee> 元素。通过执行这个规则, 将返回所有的有酬员工的 <Employee> 元素, 并且忽略在 XML 文档中遇到的其他元素。

下面的语段将执行匹配有酬员工的程序:

```
<xsl:template match="Employee[Salaried='true']">
  <Employee>
    <Name>
      <xsl:value-of select="FirstName"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="LastName"/>
    </Name>
    <Wage>
      <xsl:value-of select="Wage"/>
    </Wage>
    <Location>
      <xsl:value-of select="Location/Zip"/>
    </Location>
  </Employee>
</xsl:template>
```

上面的语句将在原始的 XML 文档中的所有有酬员工的 <Employee> 元素, 转换为所需要的新的 XML 文档。转换的具体程序代码如下:

```
private void doXMLToXMLTransform()
{
  //在屏幕上显示原始文档
  strOutput = "*** personel.xml - 原始 XML 文件 ***\r\n\r\n";
  showXMLDocument( "personel.xml" );
  //加载新文档
  //对其使用 XSL 进行转换, 并将转换生成的新文档保存至磁盘
  XPathDocument docXPath = new XPathDocument( "personel.xml" );
  XslTransform xslTransform = new XslTransform();
  XmlTextWriter writer = new XmlTextWriter( "salaried.xml", null );
  xslTransform.Load( "salariedpersonel.xsl" );
  xslTransform.Transform( docXPath, null, writer );
  writer.Close();
  strOutput += "*** salaried.xml - 转换而成的 XML 文件 ***\r\n\r\n";
  //在屏幕上显示转换而成的 XML 文件
  showXMLDocument( "salaried.xml" );
}
private void showXMLDocument( string strXMLFileName )
{
}
```



```

//加载 XML 文档
XmlDocument docDOM = new XmlDocument();
docDOM.Load( strXMLFileName );
//向文档中写内容
StringWriter writerString = new StringWriter();
XmlTextWriter writer2 = new XmlTextWriter( writerString );
writer2.Formatting = Formatting.Indented;
docDOM.WriteTo( writer2 );
writer2.Flush();
strOutput += writerString.ToString() + "\r\n\r\n";
}

```

如图 21.6 所示是运行转换后的效果。



图 21.6 XSL 转换示例

21.5.2 XSL 转换示例讲解

在 doXMLToXMLTransform()方法中，一个 XPathDocument 对象被实例化，并加载了 personnel.xml 文件。然后，实例化一个 XslTransform 对象。XslTransform 对象是一个用来在 .NET 架构中对 XML 文档执行 XSLT 转换。

当转换完成时，结果将被写入一个新的 XML 文件。一个 XmlTextWriter 对象被创建用来将文件写入磁盘，被创建的文件命名为 salaried.xml。

XslTransform 的 Load()方法从磁盘中加载了 XSL 样式表。最后，通过调用 XslTransform 类的 Transform()方法执行了文档的转换。Transform()方法以 XML 文档对象和文本写操作对象作为参数。Transform()方法的第二个参数用来传递将使用在样式表中的附加的运行参数。

说明：在本例中，因为没有运行的参数，所以传入 null。转换完成后，XmlTextWriter 对象将关闭，以结束对新的 XML 文件的写入。

showXMLDocument()方法在这个程序中是一个辅助函数，该函数的主要功能是从磁盘中读取 XML 文件，并将它显示在显示屏上。showXMLDocument()方法还演示了如何将 XmlDocument、XmlTextWriter 和 StringWriter 一起使用，而将 XML 文档转为一个字符串。

从上面的例子可以看出，进行复杂的对 XML 文档进行转换的例子实际只需要使用很少的代码。

21.5.3 XSL 转换示例二

本节需要做的就是将新生成的 XML 文档转化为 HTML 报表。下面是文件 `salariesreport.xsl` 的具体内容，是将 XML 文件转换为 HTML 文件所需要的 XSL 样式表。

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>EntegraTech Salaried Employees</title>
      </head>
      <body bgcolor="#C0C0C0">
        <h1 align="left">EntegraTech Salaried Employees</h1>
        <p></p>
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>
  <xsl:template match="Salaried">
    <h2 align="left">Seattle Office Salaried Employees</h2>
    <table border="0" width="100%">
      <xsl:apply-templates select="Employee[Location='98103']" />
    </table>
    <p></p>
    <h2 align="left">Portland Office Salaried Employees</h2>
    <table border="0" width="100%">
      <xsl:apply-templates select="Employee[Location='97206']" />
    </table>
  </xsl:template>
  <xsl:template match="Employee[Location='98103']">
    <tr>
      <td width="28%">
        <xsl:value-of select="Name" />
      </td>
      <td width="72%">
        <xsl:value-of select="Wage" />
      </td>
    </tr>
  </xsl:template>
  <xsl:template match="Employee[Location='97206']">
    <tr>
      <td width="28%">
        <xsl:value-of select="Name" />
      </td>
      <td width="72%">
        <xsl:value-of select="Wage" />
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

很显然，这个样式表主要用来生成显示在 Web 浏览器上的 HTML 文档。在样式表中，


读者可以发现与前一个样式表相似的模式匹配语句。这个样式表生成两个 HTML 表格，显示了雇员的姓名以及他们的工资信息。第一个表格显示了在黑龙江的雇员的信息，在第二个表格中显示了在北京的雇员的信息。下面的代码演示了如何使用样式表将 XML 转化成 HTML。

```
private void doXMLToHTMLTransformation()
{
    //加载 XML 文档,使用 XSL 对其进行转换,
    //将得到的 HTML 文件保存.
    XPathDocument docXPath = new XPathDocument( "salaried.xml" );
    XslTransform xslTransform = new XslTransform();
    XmlTextWriter writer = new XmlTextWriter( "salariedreport.html", null );
    xslTransform.Load( "salariedreport.xsl" );
    xslTransform.Transform( docXPath, null, writer );
    writer.Close();
    btnShowHTML.Enabled = true;
}
```

这段代码与前面的将 XML 文档转化为另一种格式的 XML 文档的代码非常相似。唯一需要注意的就是，XmlTextWriter 对象使用 HTML 文件的名称作为一个参数。当 doXMLToHTMLTransformation() 方法完成时，一个新的名为 salariedreport.html 的内部包含报表的 HTML 文件将被保存在磁盘中。

可以使用以下的代码打开 IE 浏览器，并且在上面显示报表。

```
private void btnShowHTML_Click_1(object sender, System.EventArgs e)
{
    System.Diagnostics.Process.Start("salariedreport.html");
}
```

 **技巧：**System.Diagnostics.Process 类的 Start() 方法用来执行一个程序，使用此方法，将 HTML 的文件名作为参数传入，IE 浏览器将会加载这个文件，并使其显示在显示屏上。

执行以上代码，将在当前文件夹中创建一个名为 salariedreport.html 的文件，同时在 IE 浏览器中显示该 HTML 页面。

21.6 本章总结

本章主要就在 .NET Framework 中如何使用 C# 语言对 XML 文档操作进行了讲解。在本章的开始部分，就 XML 的相关概念进行了介绍，然后简要介绍了如何使用 C# 语言对 XML 文件进行操作。最后介绍了如何使用 XML DOM 对 XML 文本进行操作、如何使用 XPath 对文本进行操作，以及如何使用 XSL 转换对 XML 文本进行操作。

21.7 实战练习

1. 在 Visual Studio 中新建一个 Windows 窗体应用程序，用该程序来管理学生的基本

情况,要求使用 XML 文档来保存具体的数据。程序应支持从 XML 文件中读入学生信息,允许用户在窗体中新增、修改、删除学生的信息。

2. 改写第 1 题的程序,用 DataSet 操作 XML 数据,同样要求程序能完成数据的加载、新增、修改、删除学生信息。

3. 在第 1 题的基础上增加查询功能,通过 XPath 查询学生信息,并显示在窗体中。

4. 新建一个 Windows 窗体应用程序,对第 1 题生成的 XML 文件进行处理,自定义一个 XSL 文件,用来将学生信息用表格形式生成 HTML 网页。

第 22 章 多线程编程

C#语言能够通过多线程实现并行代码的执行。每一个线程就是一个独立的可执行程序，可以独立地与其他线程同时执行。而多线程编程则可以在一个应用程序中实现多个任务的并列运行。在本章中，将首先介绍多线程编程的工作原理，并通过实例讲解如何创建一个线程。然后通过对线程的各个状态进行说明来讲解线程的同步以及异步。最后，将介绍 C#中常见的线程使用方法。

22.1 C#线程的主要特征

在本节中，将通过一系列的实例，简单地向读者说明 C#线程的主要特征。

一个 C#程序起始于一个由 CLR 与操作系统创建的单线程（main 线程），并且可通过创建其他的线程来实现多线程编程。

22.1.1 输出不同内容的两个线程

在本节中将创建一个简单的应用程序，在这个应用程序的主线程上创建一个新的线程“t”，用于运行重复打印字母“y”的方法。同时，又在主程序中使用主线程重复打印字母“x”。具体代码如下：

```
class ThreadTest {
    static void Main()
    {
        Thread t = new Thread (WriteY);
        //在新的线程上运行 WriteY() 函数
        t.Start();
        while (true)
        {
            //在操作平台上一一直打印 x
            Console.Write ("x");
        }
    }
    static void WriteY()
    {
        //在操作平台上一一直打印 y
        while (true)
        {
            Console.Write ("y");
        }
    }
}
```

🔔注意：在本章中，如无特别说明，所有的例子前面都需要指定输入 System.Threading 命名空间。

对上述程序进行编译并运行，得到的结果如下：

```

XXXXXXXXXXXXXXXXXYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*...

```

🔔说明：因为在 CLR 中为每一个线程设置了自己的堆，所以本地变量是可以保持彼此独立的。

22.1.2 调用同一方法的两个线程

首先将创建一个简单的应用程序，在这个应用程序中将会定义一个拥有本地变量的方法，然后同时在线程和新创建的线程中调用此方法。

```
static void Main()
{
    //在新线程上调用 Go() 函数
    new Thread (Go).Start();
    //在主线程上调用 Go() 函数
    Go();
}

static void Go() {
    //声明并使用一个本地变量- 'cycles'
    for (int cycles = 0; cycles < 5; cycles++)
    {
        Console.Write ('?');
    }
}
```

在上面的代码中，变量 `cycles` 被两个独立的线程分别创建在自己的堆中，所以在输出时，共输出了 10 个问号。

此段代码的运行结果如下:

??????????

如果当不同的线程对同一个对象的实例进行引用时，也就是说当线程共享某一数据时，会出现什么样的情况呢？通过下面的代码可以说明：

```
class ThreadTest
{
    bool done=false;
    static void Main()
    {
        //创建一个通用的实例
        ThreadTest tt = new ThreadTest();
        new Thread (tt.Go).Start();
        tt.Go();
    }
}
```




```

    }
    //注意: Go 方法现在是一个实例的方法
    void Go()
    {
        if (!done)
        {
            done = true;
            Console.WriteLine ("完成");
        }
    }
}

```

因为两个线程都同样调用了 ThreadTest 对象实例的 Go()方法, 它们同样共享了 done 变量, 这段程序的运行结果为“完成”。打印结果如下所示。

完成

 注意: “完成”并没有如代码逻辑中所表现的那样被打印两遍, 而是仅仅打印了一遍。造成这种结果的原因, 将在后面的内容中给予解释。

22.1.3 静态变量在多线程多线程中的应用

在多线程编程中, 静态变量的出现为多线程编程又提供了另一种完全不同的方法, 读者可以通过下面的例子进行体会。在下例中 done 被当成了一个静态变量来处理。

```

class ThreadTest
{
    //在不同线程间共享的静态方法
    static bool done;
    static void Main()
    {
        //启动一个线程运行 Go 方法
        new Thread (Go).Start();
        //在主线程上运行 Go 方法
        Go();
    }
    static void Go()
    {
        //如果 done 不为 true
        if (!done)
        {
            done = true;
            Console.WriteLine ("完成");
        }
    }
}

```

上例的输出是不确定的, 它有可能将“完成”打印两遍, 虽然由于计算机运算速度的提高, 运行结果很少出现这种情况。这个例子说明了线程中一个重要的概念——线程的安全性。


如果将 Go()方法中代码的执行顺序进行更改, 改为下面的代码, 那么在程序执行时, “完成”就会被打印两遍。

```
static void Go()
{
    //如果 done 不为 true
    if (!done)
    {
        Console.WriteLine ("完成");
        done = true;
    }
}
```

为什么会发生这种情况呢？原因是当一个线程执行 if 里的语句 if (!done) 时，也就是在将 done 被置为 true 时，另一个线程可能也执行了 WriteLine 操作。

可以通过当线程对公共文件进行读写时对其进行加锁处理来避免这种情况。具体的解决方案如下：

```
class ThreadSafe
{
    static bool done;
    static object locker = new object();
    static void Main()
    {
        //启动一个线程运行 Go 方法
        new Thread (Go).Start();
        //在主线程上运行 Go 方法
        Go();
    }
    static void Go()
    {
        //对资源加锁
        lock (locker)
        {
            if (!done)
            {
                Console.WriteLine ("完成");
                done = true;
            }
        }
    }
}
```


说明：当两个线程同时“锁”资源（在本例中是“locker”）时，一个线程必须等待，或者被挂起，直到这个锁资源重新可用。

在上面的代码中，它能保证在同一时刻仅有一个线程能够进入临界的程序段，“完成”只能被打印一次。在不确定的多线程语境下，当代码用这种方法被保护起来时，叫做线程安全。临时的中止和挂起在线程同步中是一个非常重要的特性。当然，等待一个可执行的锁只是线程被挂起的一个原因，另一个原因是线程希望被中断或休息一段时间。如下面的代码所示。

```
//线程将挂起 30 秒
Thread.Sleep (TimeSpan.FromSeconds (30));
```

一个线程同样也能够等到另一个线程结束时再开始，可以通过调用 Join() 方法实现。如下面的代码所示。


```
//假设 Go 是一个静态方法
Thread t = new Thread (Go);
t.Start();
//挂起指导线程结束
t.Join();
```

 注意：一个线程在被挂起时，并不消耗 CPU 资源。

22.1.4 线程调度

多线程是被 CLR 内部的线程调度所管理的，这是 CLR 代表操作系统的一个典型性功能。线程调度能够保证所有活动的线程都被安排了可行的操作时间，并且线程的等待和挂起（例如，需要一个可执行的锁或者一个用户的输入）并不消耗 CPU 的资源。

在一个单处理器计算机上，一个线程调度操控着时隙表，使每个活动线程间的操作能够快速地进行转换。比如本章中的第一个例子，重复打印 X 和 Y 的那段代码就是按照时间片来分配线程的。在 Windows XP 操作系统中，一个时间片是 10 毫秒，CLR 线程调度会选择一个比 CPU 的时间片要大得多的时间来进行线程间的转换。在一个多处理器计算机上，多线程将在最短时间内被执行，因为不同的线程将在不同的处理器上运行，但是它还是存在特定的时间片，因为操作系统需要时间来执行自己的线程，以及其他的应用程序。

22.1.5 线程和进程的关系

在一个单独的应用程序中，所有线程都被一个进程（即应用程序所运行的操作系统的运行单元）以一定的逻辑包含着。

线程与进程有一定的相似性，比如，多个进程在同一台计算机中运行时与多个线程同时运行在同一应用程序中运行时对资源的共享方法都是依存于时间片概念的。

线程与进程的主要区别在于，进程是完全独立于其他进程的，而线程则与其他同一应用程序中的线程共享存储区，这也正是线程间可以对临界资源进行共享的原因。

22.1.6 何时需要使用多线程

一个普通的多线程应用程序通常在后台运行时比较消耗时间，当主线程运行时，其他的工作线程则在后台运行。在 Windows 应用程序中，如果主线程正在进行一个长时间的操作，那么键盘和鼠标的操作将不会被执行，同时应用程序也变得不可响应。正是由于这些原因，需要将耗时多的程序分配给用户线程，这样就能够保证主线程不会进入无响应状态。

在没有 UI 界面的应用程序中，例如 Windows 服务，当一个线程需要等待从其他计算机（例如，应用服务器、数据库服务器或者客户端）返回的响应时，多线程变得尤其有意义。如果有一个工作线程来完成任务，则意味着其他的线程可以去做其他的事情。

一个 C# 应用程序可以通过两种方式变为多线程应用程序。第一，明确地创建并且运行附加的线程；第二，通过 .NET Framework 的特性来创建线程，比如后台工作线程、线程池、一个线程计时器、一个 Remoting 服务器、一个网络服务器或者一个 ASP.NET 应用程序。

在这些例子中，均会自动包含多线程编程。

22.1.7 何时不要使用多线程

多线程同样也会带来一定的局限性。最大的局限性就是会增加程序的复杂性。一个多线程程序不仅是在创建它时会带来复杂性，而且线程间的互操作也会带来很大的复杂性。正因为这个原因，多线程编程更适合那些线程间操作比较简单的应用程序。

多线程编程同样会带来在线程间如何分配资源和 CPU 的用时，以及使程序有效执行的方案等问题。尤其是当对磁盘的读写操作很多时，只有一个或两个资源来顺序执行任务会比有多个线程同时执行任务要快得多。

22.2 创建并开始一个线程

通过对上一节内容的学习，读者对线程的理论知识有了一定的了解。接下来，就开始编写代码，在程序中创建并执行线程。

22.2.1 用 Thread 类创建线程

线程可以通过使用 Thread 类的构造器进行创建，并传入一个 ThreadStart 类型的代理，用来指示执行什么方法。下面的代码演示了 ThreadStart 代理的定义。

```
public delegate void ThreadStart();
```

在线程上调用 start()方法并将其设为运行。线程连续运行直到此方法返回线程结束。在下面的例子中，演示了如何应用 C#语法来创建一个 ThreadStart 代理。

```
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread (new ThreadStart (Go));
        //在新的线程上运行 Go () 方法
        t.Start();
        //同时主函数上运行 Go () 方法
        Go();
    }
    //Go () 方法
    static void Go()
    {
        Console.WriteLine ("hello!");
    }
}
```

在这个例子中，线程 t 执行了 Go()方法，几乎在同时主线程上也调用了 Go()方法。运行结果如下所示，程序将几乎同时打印两个“hello!”。


```
hello!
hello!
```


通过 C# 代码，一个线程能够很方便地创建线程的代理。

```
static void Main()
{
    // 并不是必须使用 ThreadStart
    Thread t = new Thread (Go);
    t.Start();
    //...
}
//Go() 方法
static void Go()
{
    //...
}
```

在下面的例子中，包含着两个 ThreadStart 代理，一个将被编译器自动添加进来，而另一个则是使用匿名的方法启动线程。

```
static void Main()
{
    Thread t = new Thread (delegate() { Console.WriteLine ("hello!"); });
    t.Start();
}
```

另外，一个线程拥有一个 IsAlive 属性，当此线程执行 Start() 方法后，将被设置为 true，直到线程结束。

 注意：一个线程一旦结束，将不能再启动。

22.2.2 向 ThreadStart 传递参数

在上面的方法中，如果想更好地控制每一个线程的输出，可能就需要在每个线程中传入一个输入参数，可以通过向 Go() 方法中添加标记来实现这一点，但是这样将不能使用 ThreadStart 代理，因为它并不接受参数。此外，C# 语言定义了另一种可以接受参数的代理模式，叫做 ParameterizedThreadStart，具体定义方法如下所示。

```
public delegate void ParameterizedThreadStart (object obj);
```

则 22.2.1 节中的例子就可以更改为如下代码：

```
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread (Go);
        // == Go (true)
        // 启动线程
        t.Start (true);
        Go (false);
    }
    static void Go (object upperCase)
    {
```

```

        bool upper = (bool) upperCase;
        Console.WriteLine(upper ? "HELLO!" : "hello!");
    }
}

```

运行结果如下所示。

```

hello!
HELLO!

```

在本例中，编译器会自动加载 `ParameterizedThreadStart` 代理，因为 `Go()` 方法接受了一个对象参数。也可以这样写：

```

Thread t = new Thread (new ParameterizedThreadStart (Go));
t.Start (true);

```

使用 `ParameterizedThreadStart` 代理的特点是必须传入一个对象参数。

 **技巧：** `ParameterizedThreadStart` 代理也只需要一个对象参数。

还可以选择使用匿名的方式去调用一个普通的方法，如下所示。

```

static void Main()
{
    //启动一个新线程运行 WriteText() 方法
    Thread t = new Thread (delegate() { WriteText ("hello"); });
    t.Start();
}
static void WriteText (string text)
{
    Console.WriteLine (text);
}

```

这样做的优点是，目标方法（在这个例子中是 `WriteText()`）能够接受任意数量的参数。另外，还有一点值得注意，就是当匿名方法使用外部参数时，使用方法如下所示。

```

static void Main()
{
    string text = "开始";
    //启动一个新线程运行 WriteText() 方法
    Thread t = new Thread (delegate() { WriteText (text); });
    text = "结束";
    t.Start();
}
static void WriteText (string text)
{
    Console.WriteLine (text);
}

```

这段代码的运行结果如下所示。

```

结束

```

另一种常见的形式是向一个线程的实例的方法传递数据，而不是向线程的静态方法传递数据。实例对象的属性可以告诉线程如何去做。下面是重写的原始例子：

```

class ThreadTest
{
    bool upper;
}

```



```

static void Main()
{
    ThreadTest instance1 = new ThreadTest();
    instance1.upper = true;
    Thread t = new Thread (instance1.Go);
    t.Start();
    ThreadTest instance2 = new ThreadTest();
    // 主线程- 当 upper=false 时运行
    instance2.Go();
}
void Go()
{
    Console.WriteLine (upper ? "HELLO!" : "hello!");
}
}

```

22.2.3 给线程命名

一个线程可以通过它的 Name 属性来命名，这样在编译时会有很大的好处，也可以使用 Console.WriteLine 来打印线程的名字，Microsoft Visual Studio 将会获得一个线程的名字并且在 Debug 位置的工具条处将它显示出来。应用程序的主线程同样也可以被设置名字，在下面的例子中，主线程将会通过 CurrentThread 的静态属性被访问，代码如下：

```

class ThreadNaming
{
    static void Main()
    {
        Thread.CurrentThread.Name = "main";
        //启动一个线程运行 Go 方法
        Thread worker = new Thread (Go);
        worker.Name = "worker";
        worker.Start();
        //在主线程上运行 Go() 方法
        Go();
    }
    static void Go()
    {
        Console.WriteLine ("由" + Thread.CurrentThread.Name+"开始");
    }
}


```

运行结果如下所示。

```

由 main 开始
由 worker 开始

```

 注意：可以在任意时刻设置线程的名字。但是，当线程名称设置后，如果再试图更改它时，将会抛出一个异常。

22.2.4 C#的后台线程

默认情况下，线程都是前台线程，也就是说只要应用程序的所有线程中的任何一个还在运行，则应用程序就是被激活的。在 C#语言中同样支持后台线程。一个线程的 IsBackground 属性将管理着它的后台状态，如下面的代码所示。

```

class PriorityTest
{
    static void Main (string[] args)
    {
        Thread worker = new Thread (delegate() { Console.ReadLine(); });
        if (args.Length > 0)
        {
            //线程支持后台程序
            worker.IsBackground = true;
        }
        //启动线程
        worker.Start();
    }
}

```


如果一个程序被无参调用，则工作线程将会被默认的在前台线程中运行，并且在读入操作被执行时进行等待，直到用户按 Enter 键。

然而如果主线程退出，而一个前台线程依然是激活状态，那么应用程序将保持运行状态。如果有一个参数被传递给了主线程，工作程序将会被置于后台状态，并且应用程序在主线程结束时就会退出，并终止读操作。

22.2.5 设置线程优先级

一个线程的优先级决定了线程的执行时间，它与在同一应用程序中的其他的活动线程有关，具体范围如下所示。

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

 注意：只有在多线程同时活动时才是相关的。

将一个线程的优先级设置得高并不意味着它能够实时工作，因为它还是被应用程序的程序优先级所限制。如果要运行一个实时任务，在 System.Diagnostics 命名空间里的 Process 类的程序优先级必须设置成下面的样式：

```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;
```

ProcessPriorityClass.High 实际上是 process 的一个最高配置优先级：Realtime。设置一个程序的优先级为 Realtime，就是在指示操作系统不希望该线程被霸占。

如果一个实时应用程序有一个用户接口，它可能不适合提高进程的优先级，因为屏幕的刷新将会占用 CPU 的运行时间，降低整个计算机的速度，尤其是在用户接口比较复杂的情况下。降低主线程的优先级可以保证一个实时线程不会经常的刷新屏幕，但是并不意味着能够预防降低计算机的运行速度，因为操作系统会将可执行的 CPU 作为一个整体分配给一个进程。理想的解决方案是，让实时工作流和用户接口在拥有不同优先级的不同进程中运行，通过 Remoting 或者共享存储区域进行通信。

22.2.6 线程中的异常处理

在任何 try/catch/finally 程序块范围内，一个线程被创建的方式如下所示。


```

public static void Main()
{
    try
    {
        new Thread (Go).Start();
    }
    //捕获异常
    catch (Exception ex)
    {
        //在本例中，此代码不被运行
        Console.WriteLine ("异常!");
    }
    static void Go()
    {
        throw null;
    }
}


```

在这个例子中的“try/catch”程序段可以说是没有意义的。但是当新创建的线程被一个 `NullReferenceException` 异常所妨碍时，也就是说当一个线程有一个非独立的可执行程序的路径时，上面的代码则变得有意义。具体如下面的代码所示。

```

public static void Main()
{
    new Thread (Go).Start();
}
static void Go()
{
    try
    {
        ...
        //这个异常将在下面被捕获
        throw null;
        //...
    }
    catch (Exception ex)
    {
        //在日志中记录异常，并且启动其他线程
        //...
    }
}


```

 **注意：**从.NET 2.0 起，任何一个未处理的异常都会关闭整个应用程序，所以在开发过程中，企图忽视任何一个异常是不可能的。因此在每一个线程块中，都需要加入 try/catch 语句块，至少在整个应用程序中，为了防止应用程序由于未处理的异常而终止，需要在整个应用程序中加入 try/catch 语句块。这种方法也许会相对麻烦一些，但却是非常必要的。

在 Windows 应用程序中，通常可以使用全局异常句柄来捕获异常。具体代码如下所示。

```
//引用命名空间
using System;
using System.Threading;
using System.Windows.Forms;
static class Program
{
    //程序入口地址
    static void Main()
    {
        Application.ThreadException += HandleError;
        Application.Run (new MainForm());
    }
    //程序引发异常时发生
    static void HandleError (object sender, ThreadExceptionEventArgs e)
    {
        //在日志中记录异常，然后退出或继续执行应用程序
    }
}
```

当一个异常是由 Windows 消息（例如键盘、鼠标或者绘图消息，简单地说就是所有的典型窗体应用程序）所引起的，将会引发 `Application.ThreadException` 事件。此事件将保证程序的安全性，而所有的此类异常都会被一个处理程序所捕获。

 **说明：**由工作线程所抛出的异常是一个异常未被应用程序所捕获的最好的例子（在主线程中的代码包括窗体构造器）。

.NET Framework 提供了较低水平的所有异常的处理程序：`AppDomain.UnhandledException`。当有未处理的异常在任意线程、任意应用程序发生时，`AppDomain.UnhandledException` 均可对它们进行处理。但是其并不能提供一个很好的日志机制来记录异常，只能使应用程序不因此而中断。如果用户需要使用日志机制来记录异常，而使应用程序继续运行的时候，则可以通过自己编写异常事件处理函数来实现。

22.3 线程同步

在多线程编程时，一些敏感数据被多个线程同时访问，此时就使用同步访问技术，保证数据的完整性。

22.3.1 线程同步和协同的可用性工具

如表 22.1 和表 22.2 所示，概括出了 .NET 对于线程操作的同步和协同的可用性工具。

表 22.1 简单的阻塞方法


构造函数	目的
<code>Sleep</code>	在给出的时间区间内阻断
<code>Join</code>	等待另一个线程来结束

表 22.2 线程的同步

类 型	构造函数	目 的	进程间通信	速度
Locking Constructs	lock	保证只有一个线程可以访问资源或一段代码	不可以	快
Locking Constructs	Mutex	保证只有一个线程可以访问资源或一段代码，能够被用来在开始处阻止应用程序的多次实例化	可以	中等
	Semaphore	保证可以有多个线程访问同一资源或一段代码	可以	中等
Signaling Constructs	EventWaitHandle	让线程处于等待状态直到它收到另一个线程的信号	可以	中等
	Wait and Pulse	让线程处于等待状态直到它收到另一个线程的信号	不可以	中等
Non-Blocking Synchronization Constructs	Interlocked	无阻塞的原子操作	可以	非常快
	volatile	允许完全的无阻塞的访问独立的文件	可以	非常快

当一个线程在等待或暂停的时候，通常是由表 22.1 和表 22.2 所包含的方法所引起的，这时线程进入通常所说的被阻塞状态。线程一旦被阻塞，将会快速地释放它所占用的 CPU 的时间，并将 `WaitSleepJoin` 属性添加到它的 `ThreadState` 属性中，此线程在取消阻塞前不会被再次调度。取消阻塞通常由下面 4 种方法引发：

- ☐ 当阻塞条件被满足时；
- ☐ 当操作时间超时时；
- ☐ 当被 `Thread.Interrupt` 方法中断时；
- ☐ 当被 `Thread.Abort` 方法取消时。

 说明：当一个操作通过 `Suspend` 方法被暂停时，并不认为这个线程就中断了。

22.3.2 阻止现在的线程


通过调用 `Thread.Sleep` 方法可以在指定的时间内（或直到被打断）阻止现在的线程。具体使用方法如下所示。

```
static void Main()
{
    //放弃 CPU 的时间片
    Thread.Sleep (0);
    //将线程停止 1000 毫秒
    Thread.Sleep (1000);
    //将线程停止 1 小时
    Thread.Sleep (TimeSpan.FromHours (1));
    //将线程停止到被打断
    Thread.Sleep (Timeout.Infinite);
}
```

准确地说，线程的 `Thread.Sleep()` 方法将使线程不再占用 CPU，线程的请求也不会被再

次调度，直到规定的事件结束。`Thread.Sleep(0)`则只是停止对 CPU 的占用，以使其他排列在时间片队伍中活动的线程被执行。

`Thread` 类同时也提供了 `SpinWait` 方法，使用这个方法并不中断对 CPU 的占用，而是使 CPU 进入一个循环，一个给定的数字将决定循环的次数。每 50 次迭代的时间消耗大概是 1 微秒，当然这也取决于 CPU 的执行和加载速度。从技术角度说，`SpinWait` 方法并不是一个阻塞的方法，一个调用 `SpinWait()` 方法的线程进入阻塞状态时，线程状态并不是 `WaitSleepJoin`，其中断状态也不能被其他线程打断。`SpinWait()` 方法很少被用到，它主要用于需要在中断时继续占用资源，且中断时间非常短不需要再调用 `Sleep()` 方法来浪费 CPU 的线程转换时间时使用。

 **技巧：**通常来说，`SpinWait()` 方法只有应用在多处理器计算机上时才具有优势。

22.3.3 Joining 一个线程

使用 `Join` 方法可以阻塞进程直到另一个进程调用 `Join()` 方法，如下面的代码所示。

```
class JoinDemo
{
    //程序入口
    static void Main()
    {
        //初始化线程 t
        Thread t = new Thread (delegate() { Console.ReadLine(); });
        //启动线程
        t.Start();
        //等待，直到 t 线程结束
        t.Join();
        Console.WriteLine ("Thread t's ReadLine complete!");
    }
}
```

`Join()` 方法同样也可以接受一个以微秒为单位的时间参数或者时间间隔。如果在调用 `Join()` 方法时，此线程已经中止，则会返回 `false`。带有时间参数的 `Join()` 方法在功能上和 `Sleep()` 方法一样，也就是说，下面两行代码几乎具有相同的功能。

```
Thread.Sleep (1000);
Thread.CurrentThread.Join (1000);
```

22.4 线程安全

在前文中已经提过，通过给资源上锁可以使资源具有访问的唯一性，也就是说在一定时间内只有一个线程可以访问这段代码。本节将结合具体的例子讲解线程安全的相关知识。

22.4.1 了解线程安全

首先，考虑下面的类：


```

class ThreadUnsafe
{
    //静态变量 val1, val2
    static int val1, val2;
    //静态方法 Go()
    static void Go()
    {
        //如果 val2 不为 0
        if (val2 != 0)
        {
            Console.WriteLine (val1 / val2);
        }
        val2 = 0;
    }
}

```

这个类并不是线程安全的，如果 Go() 方法被两个线程同时调用，它可能会得到一个分母不能为 0 的错误。因为有可能存在这种情况，即当一个线程刚刚结束了 if 语句的判断正在执行 Console.WriteLine 语句的时候，val2 变量正好被另一个线程设置为 0。

可以通过 lock() 方法解决上述问题，具体代码如下所示。


```

class ThreadSafe
{
    static object locker = new object();
    static int val1, val2;
    //静态方法 Go()
    static void Go()
    {
        //通过 lock() 方法解决 Go() 方法被两个线程同时调用的问题
        lock (locker)
        {
            //如果 val2 不为 0
            if (val2 != 0)
            {
                Console.WriteLine (val1 / val2);
            }
            val2 = 0;
        }
    }
}

```


在一段时间内只有一个线程可以访问已经上锁的资源（在本例中是 locker 对象），而其他任何线程或者程序段都必须等锁被释放后才能访问此资源。如果多于一个线程包含着锁，它们将排队等待并且按照先到先得的原则对资源进行访问。有时候，可执行的锁也可被认为是保护被锁资源只能进行线性的访问，这是因为一个线程的访问不能与其他线程的访问相交叠。在本例中，锁方法保护了 Go() 方法中的逻辑，也就是保护了变量 val1 和 val2。

一个处于阻塞状态等待锁资源的线程的线程状态是 WaitSleepJoin。后面将讨论一个位于此种状态中被阻塞的资源怎样通过 Interrupt() 或者 Abort() 方法被释放，这种技术可以用来结束任何一个工作进程。

 说明：C# 的锁语句实际上就在 try-finally 语段中，调用 Monitor.Enter() 和 Monitor.Exit() 两种方法的简化版本。

在前面的例子中 Go()方法实际的运行情况，如下面的代码所示。

```
//在指定对象上获取排他锁
Monitor.Enter (locker);
try
{
    //如果 val2 不为 0
    if (val2 != 0)
    {
        Console.WriteLine (val1 / val2);
    }
    val2 = 0;
}
finally
{
    //释放指定对象上的排他锁
    Monitor.Exit (locker);
}
```

说明：如果在没有调用 Monitor.Enter()方法前就在一个对象上调用了 Monitor.Exit()方法，那么这个对象将抛出一个异常。

22.4.2 选择一个同步对象

在共享线程中任意一个可见的对象只要它是引用类型的，那么都可以用来作为线程同步的对象。同样，也可以在类的范围内对对象进行同步化，这样可以防止外部代码与已上锁对象间的无意识交互。只要遵循上面的原则，就可以加速对象的同步化进程了。如下面对 List 对象进行操作的代码所示。

```
class ThreadSafe
{
    //定义 List 类型的变量
    List <string> list = new List <string>();
    void Test()
    {
        //同一时间只有一个线程可对 list 进行操作
        lock (list)
        {
            list.Add ("第 1 项");
            //...
        }
    }
}
```

在创建一个同步化的对象时，通常会有一个专门的对象被使用（如前面例子中所提到的 locker 对象）通过它能够精确地控制锁的作用范围和作用时间。用对象本身或者此对象的类型作为同步化的对象是不被推荐的，因为它已经是同步化对象潜在的作用范围。

如下面代码所示：

```
lock (this)
{
    //...
}
```


或者:

```
lock (typeof (Widget))
{
    //...
}
```

22.4.3 使用嵌套锁

一个线程能够重复的对同一个对象上锁,既可以通过多次调用 `Monitor.Enter()` 方法,也可以通过嵌套所得语句来完成。当 `Monitor.Exit()` 语句被执行时,对象将会被解锁或最外层的锁被退出。如下面代码所示,当一个方法调用另一个方法时,一个线程只有在最开始时可以被阻塞。

```
//定义静态对象
static object x = new object();
//程序入口
static void Main()
{
    //对资源上锁
    lock (x)
    {
        Console.WriteLine ("对象被上锁");
        //调用 Nest() 方法
        Nest();
        Console.WriteLine ("对象依然被上锁");
    }
    //在此处锁被释放

    //Nest() 方法
    static void Nest()
    {
        lock (x)
        {
            //...
        }
        //释放锁,不退出
    }
}
```

22.4.4 什么时候上锁合适


在上锁时有一个基本的原则,即任何多线程的文件访问都需要在锁的作用范围内读和写。即使是最简单的例子,也必须使用这种方法。例如下面的类中, `Increment()` 方法和 `Assign()` 方法都不是线程安全的,代码如下:

```
class ThreadUnsafe
{
    static int x;
    // Increment() 方法不是线程安全的
    static void Increment()
    {
        x++;
    }
}
```

```
// Assign()方法不是线程安全的
static void Assign()
{
    x = 123;
}
}
```

而下面的类给出的则是线程安全的 Increment()方法和 Assign()方法，代码如下：

```
class ThreadUnsafe
{
    //静态变量
    static object locker = new object();
    static int x;
    //线程安全的 Increment 方法
    static void Increment()
    {
        //对资源进行加锁，同一时间只有一个线程能对 x 变量进行操作
        lock (locker)
        {
            x++;
        }
    }
    //线程安全的 Assign 方法
    static void Assign()
    {
        lock (locker)
        {
            //对资源进行加锁，同一时间只有一个线程能对 x 变量进行操作
            x = 123;
        }
    }
}
```

说明：作为一个具有可转换性的锁对象，在这种简单的情形中，也可以使用无阻塞的同步构造器。这些属于线程编程的高级特性。

如果一组变量总是在相同的锁内被读写，那么可以说变量的读写操作是具有原子性的。假设文件 x 和 y 总是在一个锁中被锁对象读取，则可以说，x 和 y 文件的访问是原子性的，因为代码段已经不能被其他操作的代码段分割了。例如，在下面的代码示例中，用户永远不会出现分母为 0 的错误异常：


```
//用户永远不会出现分母为 0 的错误异常
lock (locker)
{
    //如果 x 不为 0
    if (x != 0)
    {
        y = y / x;
    }
}
```

22.4.5 使用锁的效率考虑

锁本身是一个非常快速的操作。在无阻塞的情况下，一个锁操作通常只需要 10 纳秒

的时间,如果锁操作遇到了阻塞,因此而产生的任务转换将会消耗接近微秒级的时间长度,即使线程真正被调用才仅需要几毫秒的时间。

如果不正确的使用锁操作,很容易带来不利的影响,例如死锁现象。当有很多代码被写在锁语句中时,则会发生无数的并发操作,也就会使得其他的线程进入阻塞状态。死锁现象就是当两个线程互相等待另一个线程所占用的锁时,这两个线程都无法继续运行下去。死锁的发生通常是因为有太多的同步化对象交错在一起。而锁竞争则发生在当两个线程都同时到达了锁资源的时候。

 **注意:** 如果是一个错误的线程获得了锁资源,则程序将会被打断。

22.4.6 线程安全与 .NET Framework

线程安全是说代码里并没有不确定的多线程处理语境。线程安全通过锁可以得到很大的改善,同时也减少了线程间交互的可能性。在一些语境中,有一种线程安全的方法叫做可重入。使用此方法可以保证资源的完整性。

“锁”方法可以很好地被用来将线程不安全的代码转化成线程安全的代码。.NET Framework 就是一个很好的例子。在 .NET Framework 中几乎所有的非原始类型在被实例化时,都不是线程安全的。可是如果所有对给出对象的访问都被 lock 语句处理过时,则它们还是可以用于多线程的编程。

下例中,共有两个线程同时向一个 List 类型执行添加项的操作,然后列举出 List 中的各项。

```
class ThreadSafe
{
    static List <string> list = new List <string>();
    static void Main()
    {
        //启动线程运行 AddItems() 方法
        new Thread (AddItems).Start();
        //启动线程运行 AddItems() 方法
        new Thread (AddItems).Start();
    }
    //静态方法
    static void AddItems()
    {
        for (int i = 0; i < 100; i++)
        {
            //同一时间只有一个线程可对 list 进行操作
            lock (list)
            {
                list.Add ("项 " + list.Count);
            }
        }
        string[] items;
        //同一时间只有一个线程可对 list 进行操作
        lock (list)
        {
            items = list.ToArray();
        }
        //遍历 items 中的各项进行打印
    }
}
```


```
foreach (string s in items)
{
    Console.WriteLine (s);
}
}
```

在本例中，代码对 List 对象本身上锁，这在上例中的简单语境里是可以实现的。但是，如果有两个相关联的 List，那么就需要对一个独立的公共对象加锁。

.NET 的枚举集合在这种情况下同样也是非线程安全的。如果另一线程在枚举过程中更改了列表将会抛出一个异常。在这个例子中，与其在运行过程中对枚举类型加锁，不如首先将各项复制到一个数组中。这样可以避免由于长时间的持有锁而造成在枚举过程中的时间消耗。

无论 List 类是否真的是线程安全的，上面的语句都不是线程安全的。整个 if 语段应该被一个 lock 语句包围在里面，这样，可以防止向 List 添加项的操作和测试 List 所包含的项的操作之间发生冲突。同样，这种锁可以用在当代码对 List 进行操作时的任何地方。比如，在下面的语句中也需要同样的 lock 语句将其包装在里面，以保证不与前面的代码产生冲突。

```
myList.Clear();
```

 注意：在编程时，需要对所有的非线程安全的集合类进行加锁操作。

22.5 用中断和取消提前释放线程

一个线程段可以通过以下两种方法提前释放：

- ☐ 通过线程中断；
- ☐ 通过线程取消。

这两种方法都必须通过另外一个活动的线程引发，因为一个等待的线程在处于阻塞状态时是不能做任何事的。

22.5.1 中断线程

在一个阻塞的线程上强行调用 Interrupt() 方法，将会抛出一个 ThreadInterruptedException 异常。具体情况如下面的代码所示。

```
class Program
{
    //程序入口
    static void Main()
    {
        //定义新线程
        Thread t = new Thread (thead);
        //启动线程
        t.Start();
        //中断线程
        t.Interrupt();
    }
}
```



```
//定义线程 thead
static void thead()
{
    //线程等待
    try
    {
        Thread.Sleep (Timeout.Infinite);
    }
    //捕获异常
    catch (ThreadInterruptedException)
    {
        Console.Write ("强制执行");
    }
    Console.WriteLine ("完成!");
}
}
```

上面代码的运行结果是：

强制执行完成！

中断一个线程只是将它从现在的等待中释放出来，并不会使线程结束（除非没有捕获 `ThreadInterruptedException` 异常）。

如果中断方法是被一个没有陷入阻塞状态的线程所调用，那么线程将会继续执行，直到下一次阻断发生，抛出 `ThreadInterruptedException` 异常。也就是说，下面代码中，`if` 语句所进行的判断是没有意义的。

```
//if 语句所进行的判断是没有意义的
if ((worker.ThreadState & ThreadState.WaitSleepJoin) > 0)
{
    //中断进程
    worker.Interrupt();
}
```

22.5.2 取消线程

一个处于阻断状态的线程也可以通过使用 `Abort()` 方法而强行释放资源。这样做的效果和调用 `Interrupt()` 方法唯一的不同之处在于，抛出的异常不再是 `ThreadInterruptedException`，而是 `ThreadAbortException`。此外，异常在 `catch` 语段的最后也将被抛出，除非在 `catch` 语段中再次调用 `Thread.ResetAbort()` 方法。此时，线程的状态将被置于 `AbortRequested`。

`Interrupt()` 方法和 `Abort()` 方法中的最大不同在于，当一个线程调用它们时，如果线程当时不处于阻塞状态，那么 `Interrupt()` 方法会等到下一次阻塞发生时再执行操作，而 `Abort()` 方法则在线程执行到的地方抛出一个异常。

 注意：在没有阻塞的线程中调用 `Abort()` 方法将会引发很严重的后果。

22.6 线程有哪些状态

当线程执行时，代码可以通过对线程的 `ThreadState` 属性进行读取来获取线程的当前状

态。如图 22.1 所示，列举了一个线程的所有状态，以及它们的触发方法。

- ❑ Unstarted 是尚未对线程调用 `Thead.Start()` 方法时所处的状态。
- ❑ WaitSleepJoin 是在对线程调用 `Thead.Sleep()` 方法或 `Thead.Join()` 方法后，线程被阻塞时处于的状态。
- ❑ Run 是在对线程调用了 `Thead.Start()` 方法，线程已经启动且没有阻塞时处于的状态。
- ❑ Stopped 是线程停止时处于的状态。
- ❑ AbortedRequested 是在对线程调用 `Thead.Abort()` 方法后，但尚未进入 `Stopped` 状态前的状态。
- ❑ Aborted 是线程处于 `Stopped` 的状态。

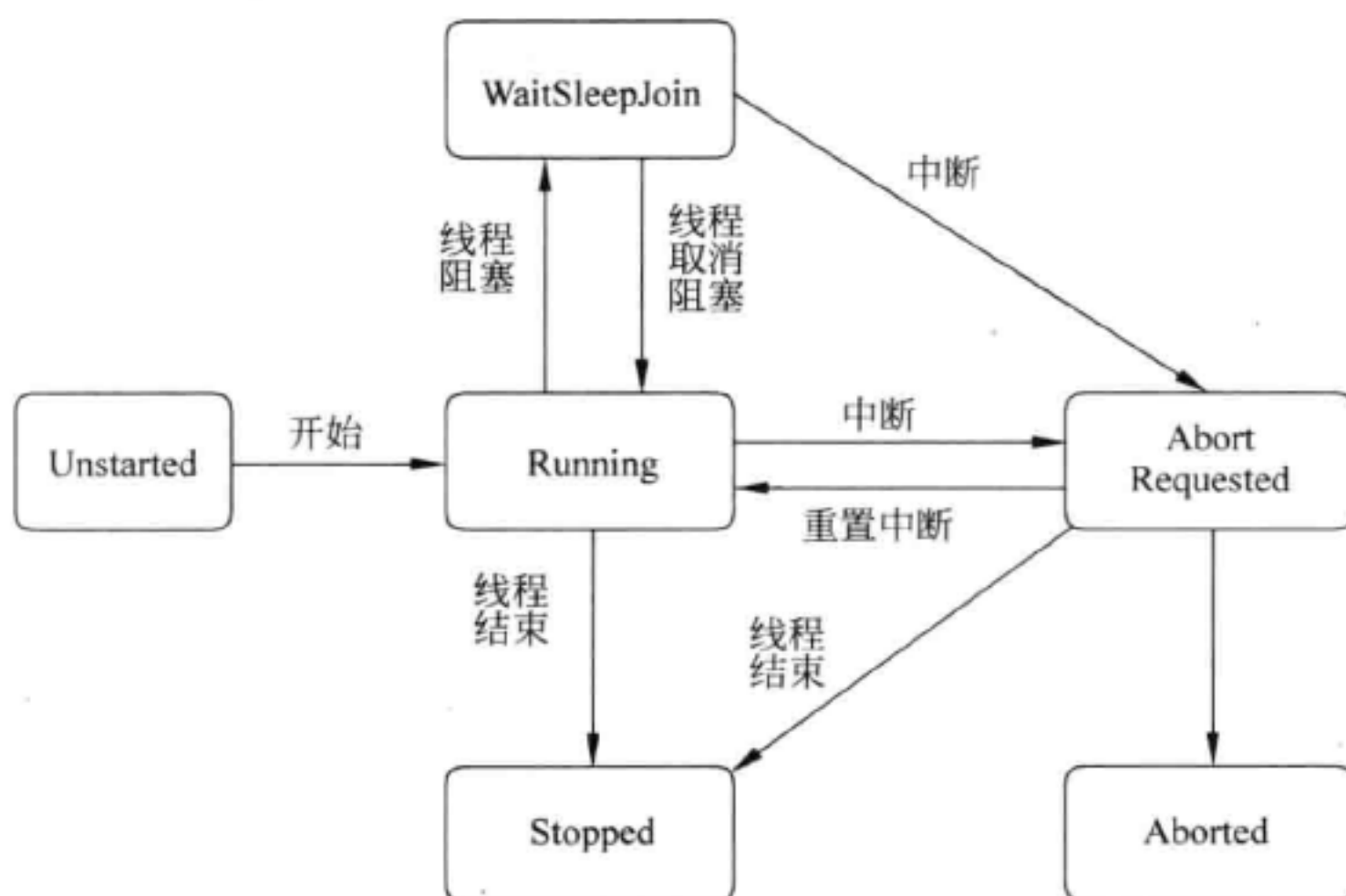



图 22.1 线程状态图

22.7 等待处理

`lock` 语句是线程同步的构造函数的一个例子。`lock` 语句适合在对某一资源或某段代码进行唯一性访问执行访问操作时用到。但是，除了此任务外，还有一些其他的同步化任务需要处理，例如，将某一重要的处于等待状态的线程置于开始执行任务阶段。

WIN32 API 拥有丰富的同步化构造函数，这些构造函数都可以通过 `EventWaitHandle`、`Mutex`（互斥量）和 `Semaphore`（信号量）应用到 .NET Framework 中。这些方法有些还非常有效，例如 `Mutex` 的效率比使用 `lock` 语句提升了近两倍，`EventWaitHandle` 方法也提供了唯一的信号量功能。

 说明：`EventWaitHandle`、`Mutex`（互斥量）和 `Semaphore`（信号量）这 3 个类都是基于 `WaitHandle` 基类，虽然它们的行为截然不同，但是它们之间却有一个共同点，就是都可以被再次命名，都可以对操作系统的进程工作，而不仅仅是跨当前进程工作。

`EventWaitHandle` 提供了两个基类：`AutoResetEvent` 和 `ManualResetEvent`。这两个类都

是从它们的基类中获得它们的全部功能，唯一不同的是它们的构造函数的参数不同。从运行效果来看，Wait 操作的相关方法运行时间都在微秒范围内。

22.7.1 了解自动设置方法

AutoResetEvent 类可以很形象地被解释为“十字转门门票”，它就像公园门口所见的十字旋转门的门票一样，每塞进一张门票就有一人也仅有一人可以通过。该类名中的 Auto 在英语里是自动的意思，在这里可以理解成在一个人塞入门票时，此旋转门都可以自动地打开，而在另一个人通过前旋转门关闭并重置。一个线程的等待或阻塞就类似于在旋转门前等待它再次开启。如果有一系列的线程在等待旋转门的开启就会形成一个队列。进入旋转门的门票可以是来自队列中的任何一个线程。也就是说，任何访问 AutoResetEvent 对象的线程（没上锁）都可以调用 Set() 方法来释放被阻塞的线程。

如果在没有线程等待的时候调用了 Set() 方法，那么句柄将一直等到有线程进入等待旋转门开启的状态。这种方法可以帮助避免一个线程已经进入旋转门，而另一个线程却塞入了门票的现象发生。然而，如果在一个没有线程等候的旋转门前重复调用 Set() 方法是不被允许的。

当一个线程处于等待状态时，可以设置一个时间限制的参数，如果等待的时间超时，线程的等待操作将结束，并返回一个值为 false 的 bool 变量。也可以使用 Reset() 方法来关闭本应是开启状态的旋转门。


AutoResetEvent 可以通过两个途径被创建，第一种方法是使用构造函数，代码如下：

```
EventWaitHandle wh = new AutoResetEvent (false);
```

如果上面代码中的 bool 变量值为 true 时，Set() 方法的句柄将在构造函数后被快速地自动调用。另一种方法是通过对基类 EventWaitHandle 的实例化来完成，如下所示。

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.Auto);
```

EventWaitHandle 同样允许通过使用 EventResetMode.Manual 参数创建一个 ManualResetEvent 的对象。

说明：一个线程一旦不再被需要，就可以通过在 Wait 句柄上调用 Close() 方法来释放操作系统的资源。然而，如果一个 Wait 句柄被用在一个应用程序的生命周期中，这一步则可以被省略，因为 Wait 句柄会在此应用程序的作用范围内自动被调用。

22.7.2 自动设置方法示例一

在下面的例子中，一个线程将被置于等待状态，直到下一个线程的信号量，代码如下：

```
class BasicWaitHandle
{
    static EventWaitHandle wh = new AutoResetEvent (false);
    static void Main()
    {
```

```

    new Thread (Waiter).Start();
    //等待 1000 毫秒...
    Thread.Sleep (1000);
    //唤醒线程
    wh.Set();
}
//静态方法 Waiter()
static void Waiter()
{
    Console.WriteLine ("等待...");
    //等待通知
    wh.WaitOne();
    Console.WriteLine ("通知");
}
}

```

运行结果如下所示，当打印出“等待...”后，要经过一秒钟的停顿才会打印出“通知”。

```

等待...
通知

```

EventWaitHandle 的构造函数同样允许对一个创建的 EventWaitHandle 对象重命名，如下面的代码所示。

```

EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.Auto,
    "MyCompany.MyApp.SomeName");

```

如果希望在后台执行任务，而不是在每一次接受任务时都显式地创建一个新线程，可以通过单线程的不断循环来达到这一要求，即“等待一个任务，执行一个任务，等待下一个任务”。这也是一个常见的多线程的解决方案。通过使用这种线性执行任务的方法，可以消除潜在的多线程间的交互和资源的浪费。

22.7.3 自动设置方法示例二

当一个新任务产生时，如果当前的工作线程却还在执行前一个任务，应该如何解决呢？如果在这里将新的任务阻塞，直至前一个任务执行完毕，那么这个系统需要两个 AutoResetEvent 对象：一个是“准备” AutoResetEvent，由已经准备好的任务进行设置；另一个是“执行” AutoResetEvent，由一个新的任务对线程的调用来设置。

下例中，将会有有一个简单的字符串用于描述任务，代码如下：

```

class AcknowledgedWaitHandle
{
    static EventWaitHandle ready = new AutoResetEvent (false);
    static EventWaitHandle go = new AutoResetEvent (false);
    // volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因
    //素更改
    static volatile string task;
    static void Main()
    {
        new Thread (Work).Start();
        //指定 5 个工作任务
        for (int i = 1; i <= 5; i++)
        {

```



```

        //在线程准备好前等待
        ready.WaitOne();
        //分配任务
        task = "a".PadRight(i, 'b');
        //执行任务
        go.Set();
    }
    //将任务置空，以通知工作线程结束
    ready.WaitOne();
    task = null;
    go.Set();
}
//静态方法 Work()
static void Work()
{
    //条件永远成立
    while (true)
    {
        //指示已经准备好
        ready.Set();
        //等待开始。
        go.WaitOne();
        //退出
        if (task == null)
        {
            return;
        }
        Console.WriteLine (task);
    }
}
}

```

运行结果如下所示。

```


ab
abb
abbb
abbbb

```

在上面的例子中，通过将任务设置为空来结束线程。

 **技巧：**也可以通过在工作线程上调用 `Interrupt()` 方法或者 `Abort()` 方法来结束线程。

另一种多线程的解决办法是让一个后台程序负责在队列中处理任务。这种方法可以叫做生产者/消费者队列，生产者将任务入队，消费者将任务出队分配给工作线程。这个方法和上一个方法很像，只是如果当工作线程目前为忙碌状态时，调用者不会进入阻塞模式。

 **说明：**在生产者/消费者队列中，可以由多个消费者被创建——每一个消费者都服务于同样的队列，但是却对应不同的线程。这是一个利用多处理器的很好的方法，但是仍要限制工作线程的数量，以避免大量线程的同时并发。

22.7.4 自动设置方法示例三

在下例中，一个单独的 `AutoResetEvent` 对象被用来作为唯一的工作线程，此线程在没

有任务时（即队列 0 为空时）进入等待状态。一个普通的集合类被用来作为队列，为了保证线程安全被一个锁文件保护着。当队列为空时，工作线程终止。代码如下：

```
//引用命名空间
using System;
using System.Threading;
using System.Collections.Generic;
//定义 ProducerConsumerQueue 类继承自 IDisposable
class ProducerConsumerQueue : IDisposable
{
    EventWaitHandle wh = new AutoResetEvent (false);
    Thread worker;
    object locker = new object();
    Queue<string> tasks = new Queue<string>();
    //构造函数
    public ProducerConsumerQueue()
    {
        //定义新线程
        worker = new Thread (Work);
        //启动新线程
        worker.Start();
    }
    //定义方法 EnqueueTask()
    public void EnqueueTask (string task)
    {
        lock (locker)
        {
            tasks.Enqueue (task);
        }
        wh.Set();
    }
    public void Dispose()
    {
        //标志消费者退出。
        EnqueueTask (null);
        //等待消费者线程结束。
        worker.Join();
        // 释放操作系统资源
        wh.Close();
    }
    void Work()
    {
        //条件永远成立
        while (true)
        {
            string task = null;
            //对资源上锁
            lock (locker)
            {
                //如果 tasks 中元素的数量大于 0
                if (tasks.Count > 0)
                {
                    task = tasks.Dequeue();
                    if (task == null)
                    {
                        return;
                    }
                }
            }
        }
    }
}
```



```

        //如果 task 不为 null
        if (task != null)
        {
            Console.WriteLine ("运行的任务: " + task);
            //线程挂起
            Thread.Sleep (1000);
        }
        else
        //没有任务
        {
            wh.WaitOne();
        }
    }
}

class Test
{
    //程序入口
    static void Main()
    {
        //过了这个空间, q 对象就释放
        using (ProducerConsumerQueue q = new ProducerConsumerQueue())
        {
            q.EnqueueTask ("开始");
            for (int i = 0; i < 10; i++)
            {
                q.EnqueueTask ("任务" + i);
            }
            q.EnqueueTask ("结束!");
        }
    }
}


```

程序的运行结果如下所示。

```

运行的任务: 开始
运行的任务: 任务 0
运行的任务: 任务 1
运行的任务: 任务 2
运行的任务: 任务 3
...
...
运行的任务: 任务 9
结束!

```

说明: 在上例中当 ProducerConsumerQueue 队列被注销时, Wait 句柄也会被关掉。因此可以在应用程序中创建很多这个类的实例。

22.7.5 ManualResetEvent 类控制多个线程

ManualResetEvent 和 AutoResetEvent 类有很大的相似性, 它们的区别是当线程调用了 WaitOne() 方法后, AutoResetEvent 类会自动将信号置于不发送状态, 而 ManualResetEvent

类不能自动对信号进行设置，线程也只能继续处于等待的状态。因此 `AutoResetEvent` 类一次只能唤醒一个线程，而 `ManualResetEvent` 类则可以同时控制多个线程。

22.7.6 跨线程的互斥量

`Mutex`（互斥量）提供了与 C# 的锁语句相类似的功能。`Mutex` 的优势在于它可以跨线程工作，也就是说，`Mutex` 提供一个更为复杂，使用面更广泛的锁。

通过使用 `Mutex` 类，`WaitOne()` 方法包含了一个可执行的锁，当出现资源抢夺时，线程会进入阻塞状态，可执行的锁可使用 `ReleaseMutex()` 方法进行释放。

 **注意：**与 C# 的锁语句相一致，`Mutex` 也只能被包含它的线程释放。

跨线程的 `Mutex` 的使用方法能够保证在程序中只有一个实例在运行。下面是它的实现方法，代码如下：

```
class OneAtATimePlease
{
    //对程序进行唯一性命名
    static Mutex mutex = new Mutex (false, "OneAtATimeDemo");
    static void Main()
    {
        //如果发生冲突则等待 5 秒钟
        //假如有程序的另一个实例正在运行，则关闭它
        if (!mutex.WaitOne (TimeSpan.FromSeconds (5), false))
        {
            Console.WriteLine ("应用程序的另一个实例正在运行!");
            return;
        }
        try
        {
            Console.WriteLine ("运行- 按 Enter 键退出");
            Console.ReadLine();
        }
        finally
        {
            mutex.ReleaseMutex();
        }
    }
}
```

本例应用程序的优点是，如果该程序终止时并没有释放 `Mutex`，那么 CLR 将会自动对其进行释放。

22.7.7 使用信号量

`Semaphore`（信号量）与互斥量不同，它有着一定的容量。`Semaphore` 就好像一个房间，一旦房间已经装满，当有更多的人想要进入时，就会在外面排成一个长队。每离开一个人，则在队伍最前面的人就可以进入这个房间。`Semaphore` 构造函数至少需要两个参数，即房间里当前可用的空间和房间的总容量。

下例中，将有 10 个线程来执行一个循环，而 `Sleep` 语句则在中间。`Semaphore` 能够保

证不多于 3 个线程可以一起执行 Sleep 语句，代码如下：

```
class SemaphoreTest
{
    //可用空间为 3
    //总容量是 3
    static Semaphore s = new Semaphore (3, 3);
    static void Main()
    {
        for (int i = 0; i < 10; i++)
        {
            //在新线程中运行静态方法 Go()
            new Thread (Go).Start();
        }
    }
    //静态方法 Go()
    static void Go()
    {
        while (true)
        {
            s.WaitOne();
            // 只有 3 个线程可以到达
            Thread.Sleep (100);
            s.Release();
        }
    }
}
```

22.7.8 使用 WaitAny、WaitAll 和 SignalAndWait 方法

除了 Set()和 WaitOne()方法外，在 WaitHandle 类中还有一些静态方法来解决更复杂的同步问题。WaitAny()、WaitAll()和 SignalAndWait()方法使多个等待句柄的等待过程变得容易实现。

SignalAndWait()方法是其中最有用的方法，它在一个 Wait 句柄里调用 WaitOne()方法，而在另一个句柄里调用 Set()方法，每一个句柄都是一个原子操作。可以在一对 EventWaitHandle 上使用这个方法创建两个线程，那么这两个线程将在同一时间到达同一点上。

第一线程如下：

```
WaitHandle.SignalAndWait (wh1, wh2);
```

而第二个线程如下：

```
WaitHandle.SignalAndWait (wh2, wh1);
```

WaitHandle.WaitAny()方法等待 Wait 句柄中的任何一个句柄即可继续进行；WaitHandle.WaitAll()方法则要等待所有给出的句柄才能继续进行。使用旋转门来举例说明，假设有很多个旋转门是同步运行的，WaitAny()方法类似于只要有一个人到达旋转门，并且塞入门票后旋转门就会全部同时打开，让其通过。而 WaitAll()方法则是需要等待所有的人都到达旋转门，塞入门票后旋转门才会同时打开，让其通过。

 注意：AutoResetEvent 或者 ManualResetEvent 也能实现上述的这种功能。

22.8 同步性作用域

比起手动上锁，还有一个方法可以对资源进行显式上锁。如果对象从 `ContextBoundObject` 类派生，并将其设置为 `Synchronization` 属性，那么可以指示 CLR 对此对象自动上锁。具体代码如下：

```
//引用的命名空间
using System;
using System.Threading;
using System.Runtime.Remoting.Contexts;
//将属性设置为 Synchronization
[Synchronization]
//继承自 ContextBoundObject
public class AutoLock : ContextBoundObject
{
    public void Demo()
    {
        Console.Write ("开始...");
        //在此处不可以抢占资源，让线程等待 1 秒钟
        Thread.Sleep (1000);
        //自动上锁
        Console.WriteLine ("结束");
    }
}

public class Test
{
    //程序入口地址
    public static void Main()
    {
        AutoLock safeInstance = new AutoLock();
        // 调用 Demo 函数
        new Thread (safeInstance.Demo).Start();
        // 执行 3 次
        new Thread (safeInstance.Demo).Start();
        // 并发。
        safeInstance.Demo();
    }
}
```

运行结果如下：

```
开始... 结束
开始... 结束
开始... 结束
```

CLR 能够保证在线程安全的实例中，同一时间内只有一个线程可以执行代码。它通过创建一个单线程对象来执行代码，并且禁止其他方法对线程安全的实例的方法和属性进行调用。在这个例子里，锁的范围是被声明的作用域，即这个线程安全的对象。

`Synchronization` 属性的命名空间是 `System.Runtime.Remoting.Contexts`。一个 `ContextBound` 对象可以被想象成是一个远程对象——也就是说所有对线程安全的实例的访问都是被拦截的。为了使这种拦截有效，可以通过将 `AutoLock` 类实例化，那么 CLR 将会自动返

回一个代理——一个拥有 AutoLock 所有属性和方法的对象，这个代理将起到中介的作用，可以通过这个中介对对象自动上锁。

因此，下面代码将得到与上面代码一样的运行结果。

```
//将属性设置为 Synchronization
[Synchronization]
//继承自 ContextBoundObject
public class AutoLock : ContextBoundObject
{
    public void Demo()
    {
        Console.Write ("开始...");
        //线程挂起
        Thread.Sleep (1000);
        Console.WriteLine ("结束");
    }
    public void Test()
    {
        //在新线程启动 Demo 方法
        new Thread (Demo).Start();
        new Thread (Demo).Start();
        new Thread (Demo).Start();
        Console.ReadLine();
    }
}
//程序入口
public static void Main()
{
    new AutoLock().Test();
}
```

 **技巧：**在这段代码中添加了 Console.ReadLine 语句，这样程序在此处可以暂停执行。

因为在同一时间内，一个类的对象里只有一个线程可以执行代码，那么这 3 个新创建的线程在 Demo()方法中将会保持阻塞状态，直到 Test()方法结束（Test()方法的结束需要执行完 ReadLine 语句）。因此，在按 Enter 键后，将得到与前段程序一样的结果。

但是，在上面的程序中，并没有解决前面所描述的问题，即如果 AutoLock 是一个集合类的时候该如何解决这一问题。比如，当使用下面的代码时，就需要使用 lock 语句对此语段加锁，除非此段代码使用 ContextBoundObject 类对其本身进行同步化。

```
if (safeInstance.Count > 0) safeInstance.RemoveAt (0);
```

如果类 SynchronizedA 的对象实例化类 SynchronizedB 的对象，而类 SynchronizedB 的声明如下所示，那么它们将拥有独立的同步化空间。

```
[Synchronization (SynchronizationAttribute.REQUIRES_NEW)]
public class SynchronizedB : ContextBoundObject
{
    ...
}
```

对象同步化的作用空间越大，对象就越利于管理，但是却减少了产生有效并发的机会。在其他范围的结尾，独立的同步化容易引发死锁。如下面的代码所示。

```
//将属性设置为 Synchronization
[Synchronization]
```

```
//继承自 ContextBoundObject
public class DeadLock : ContextBoundObject
{
    public DeadLock Other;
    //Demo()方法
    public void Demo()
    {
        //程序挂起
        Thread.Sleep (1000);
        Other.Hello();
    }
    //Hello()方法
    void Hello()
    {
        Console.WriteLine ("hello");
    }
}
public class Test
{
    //程序入口
    static void Main()
    {
        Deadlock dead1 = new Deadlock();
        Deadlock dead2 = new Deadlock();
        dead1.Other = dead2;
        dead2.Other = dead1;
        //在新线程中启动 Demo()方法
        new Thread (dead1.Demo).Start();
        dead2.Demo();
    }
}
```

在上面的代码中，每一个 Deadlock 类的实例都在一个非同步化的类 Test 中被创建，而每一个被创建的对象将到达它们自己的同步化范围，因此，将会被自己锁定。当两个对象互相调用对方时，死锁就发生了。

22.9 套间线程

22.9.1 什么是套间线程


套间(Apartment)线程是一个保证原子线程安全的规则，它与 COM(Components Object Model) 组件紧密关联。而.NET 在很大程度上打破了这种传统的线程模型，它只有在程序需要与 API 进行互操作时才会出现。套间线程与 Windows 窗体有很大的关联，因为大多数 Windows 窗体都使用或包含了大量的标准 Win32 API。

一个套间在逻辑上就是一个线程的容器。共有两种类型的套间模型——“单线程套间模型”和“多线程套间模型”。一个单线程套间只包含一个线程，而多线程套间则可以包含很多个线程。单线程套间模型要比多线程套间模型具有更强的交互性，且比较常用。

可以将套间想象成一个图书馆，每一本书代表了一个对象。在这个图书馆中是不可以

借出图书的，那么这些图书将一直留在图书馆中。而图书馆里的人则代表了一个线程。在一个同步化环境的图书馆中是允许任何人进入的，但是每次却只可以进入一个人，因此图书馆外就会有一排等待进入图书馆的队伍。

套间图书馆里还会有一些常驻人员——每个单线程的图书馆有一个图书管理员，而多线程的图书馆则有一组管理员。

 **说明：**如同套间可以包含线程一样，套间也可以包含对象。当一个对象被一个套间创建时，那么它只在这个套间里有效。这与一个对象包含在.NET 同步化范围内很类似，除非一个同步化的作用范围不包含或者还拥有线程。任何线程都可以在任意一个同步化的环境里调用一个对象。但是在套间中创建的对象只能调用套间中的线程。

22.9.2 使用套间模型

一个.NET 线程将被自动地加入到一个套间中，如果没有特别说明，它将被加入多线程套间模型。如需要加入到单线程套间模型，可使用如下代码：

```
Thread t = new Thread (...);
t.SetApartmentState (ApartmentState.STA);
```

也可以使用 `STAThread` 属性将一个单线程套间加载到一个主线程中，具体代码如下所示。

```
class Program
{
    [STAThread]
    static void Main()
    {
        ...
    }
}
```

套间在执行.NET 代码时不会起到任何作用。也就是说，在单线程套间模型中也可以有两个线程访问同一个对象的同一个方法，且不会产生自动的上锁和封送。

22.9.3 用 `Control.Invoke` 方法进行跨线程调用

在一个多线程的 Windows 窗体应用程序中，在任意线程中调用控件的方法或属性是不被允许的，除非是创建这个控件的线程。所有的跨线程调用都将通过 `Control.Invoke()` 或者 `Control.BeginInvoke()` 方法被精确地封送到创建这个控件的线程（通常是主线程）中去。

一个优秀的解决方案通常是通过 `BackgroundWorker` 组件来处理 Windows 窗体或 WPF 应用程序的工作线程。这个类能够在需要的时候自动地调用 `Control.Invoke` 或者 `Control.BeginInvoke` 方法。

22.10 管理工作线程的 `BackgroundWorker` 组件


使用 `BackgroundWorker` 组件可以不用显式定义线程，而使用多线程的一种方法。通过

BackgroundWorker 组件可以在单独的专用线程上运行一些耗时的操作（如下载和数据库事务），以避免由于长时间运行时导致用户界面出现假死的状态。

22.10.1 BackgroundWorker 的特征

BackgroundWorker 组件是一个 System.ComponentModel 命名空间中用于管理工作线程的帮助类。它具有以下特征：

- ☐ 它具有一个“取消”标记，用于在一个工作线程结束却没有调用 Abort()方法时进行标记。
- ☐ 有一个标准的协议用于报告进度、完成和取消状态。
- ☐ BackgroundWorker 组件可以以拖曳的方式添加到 Visual Studio 窗体设计器中。
- ☐ 可以对工作线程进行异常处理。
- ☐ 在工作线程运行过程中或者结束时可以对 Windows 窗体和 WPF 控件进行更新。

 说明：BackgroundWorker 组件的最后两个特性非常有用，它意味着在开发人员进行开发时不需要再对语句段设置 try/catch 语句，并且能自动对 Windows 窗体和 WPF 控件进行更新，而不需要再调用 Control.Invoke()方法。

22.10.2 BackgroundWorker 组件编程示例一

BackgroundWorker 组件使用了线程池技术，也就是说将线程池化，以防止它们每当在新任务来临时频繁创建造成资源的消耗。这意味着在 BackgroundWorker 线程中永远不应该调用 Abort()方法。


使用 BackgroundWorker 组件的最少步骤是：

- (1) 实例化 BackgroundWorker，并且处理 DoWork 事件。
- (2) 调用 RunWorkerAsync()方法。

下面是使用 BackgroundWorker 组件的具体例子，代码如下：

```
class Program
{
    static BackgroundWorker bw = new BackgroundWorker();
    //程序入口
    static void Main()
    {
        bw.DoWork += bw_DoWork;
        bw.RunWorkerAsync ("向工作线程传递消息");
        Console.ReadLine();
    }
    static void bw_DoWork (object sender, DoWorkEventArgs e)
    {
        // 在工作线程中被调用
        //打印出“向工作线程传递消息”
        Console.WriteLine (e.Argument);
        // 执行任务...
    }
    //...
}
```


BackgroundWorker 组件同样提供了一个 RunWorkerCompleted 事件, 此事件在 DoWork 事件执行时被引发。RunWorkerCompleted 事件的执行并不是强制性的, 但是通常情况下, 可以按此顺序进行, 并且可以捕获在 DoWork 事件中抛出的异常。

 **技巧:** 在 RunWorkerCompleted 中的代码可以不需要封送而更新 Windows 窗体和 WPF 控件, 但在 DoWork 中的代码却不具备此项功能。

22.10.3 BackgroundWorker 组件编程示例二

在程序中添加进程报告的支持共分为以下几个步骤:

- (1) 将 WorkerReportsProgress 属性设置为 true。
- (2) 在 DoWork 事件处理函数中调用 ProgressChanged 事件。
- (3) 处理 ProgressChanged 事件, 查询此事件参数的 ProgressPercentage 属性。
- (4) 在 ProgressChanged 事件处理函数中也可以像使用 RunWorkerCompleted 一样很轻松地处理控件用户接口。

这种方法尤其适用于开发人员需要用 Progressbar 控件显示程序进度时。具体的实现步骤为:

- (1) 将 WorkerSupportsCancellation 属性设置为 true。
- (2) 在 DoWork 事件中周期性的检查 CancellationPending 属性, 如果该属性值为 true, 则将此事件的 Cancel 属性设置为 true 并返回。
- (3) 调用 CancelAsync() 方法来请求撤销。

在下面的代码中, 将对上面所叙述的特性进行具体的执行。

```
//引用命名空间
using System;
using System.Threading;
using System.ComponentModel;
// Program 类
class Program
{
    static BackgroundWorker bw;
    //主程序入口
    static void Main()
    {
        bw = new BackgroundWorker();
        //支持进度报告
        bw.WorkerReportsProgress = true;
        bw.WorkerSupportsCancellation = true;
        bw.DoWork += bw_DoWork;
        bw.ProgressChanged += bw_ProgressChanged;
        bw.RunWorkerCompleted += bw_RunWorkerCompleted;
        //打印提示信息
        bw.RunWorkerAsync ("开始");
        Console.WriteLine ("在 5 秒钟后按 Enter 键取消");
        Console.ReadLine();
        if (bw.IsBusy)
        {
            bw.CancelAsync();
        }
    }
}
```

```

    Console.ReadLine();
}
//在bw工作时被启动
static void bw_DoWork (object sender, DoWorkEventArgs e)
{
    for (int i = 0; i <= 100; i += 20)
    {
        if (bw.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
        //报告进程
        bw.ReportProgress (i);
        //程序挂起
        Thread.Sleep (1000);
    }
    // 将结果传给 RunWorkerCompleted
    e.Result = 123;
}
static void bw_RunWorkerCompleted (object sender,
    RunWorkerCompletedEventArgs e)
{
    //如果操作被取消
    if (e.Cancelled)
    {
        Console.WriteLine ("取消操作!");
    }
    //如果操作出现异常
    else if (e.Error != null)
    {
        Console.WriteLine ("异常: " + e.Error.ToString());
    }
    else
    {
        //打印提示信息
        Console.WriteLine ("完成 - " + e.Result);
    }
}
static void bw_ProgressChanged (object sender, ProgressChangedEventArgs e)
{
    //打印提示信息
    Console.WriteLine ("到达" + e.ProgressPercentage + "%");
}
}

```

运行结果如下所示。

```

在 5 秒钟后按 Enter 键取消
到达 0%
到达 20%
到达 40%
到达 60%
到达 80%
到达 100%
完成 - 123
在 5 秒钟后按 Enter 键取消
到达 0%

```


到达 20%
 到达 40%
 取消操作!

22.10.4 BackgroundWorker 组件编程示例三

BackgroundWorker 组件并没有封装和提供一个 OnDoWork() 虚方法, 使开发人员可以对此方法进行重载。当要执行一个运行很长时间的方法时, 开发人员可以写一个 BackgroundWorker 类的子类, 在异步的执行任务前进行配置, 只需要对处理事件的方法进行重写即可。

例如, 假如开发人员要写一个名为 GetFinancialTotals() 的很消耗时间的方法, 如下所示。

```
public class Client
{
    // GetFinancialTotals() 方法
    Dictionary <string,int> GetFinancialTotals (int foo, int bar)
    {
        ...
    }
    ...
}
```

则可以对其进行重构, 得到如下所示的代码。


```
public class Client
{
    // GetFinancialTotals() 方法
    public FinancialWorker GetFinancialTotalsBackground (int foo, int bar)
    {
        // 实例化 FinancialWorker 类
        return new FinancialWorker (foo, bar);
    }
}
//继承自 BackgroundWorker
public class FinancialWorker : BackgroundWorker
{
    public Dictionary <string,int> Result;
    public volatile int Foo, Bar;
    //构造函数一
    public FinancialWorker()
    {
        WorkerReportsProgress = true;
        WorkerSupportsCancellation = true;
    }
    //构造函数二
    public FinancialWorker (int foo, int bar) : this()
    {
        this.Foo = foo;
        this.Bar = bar;
    }
    //对 OnDoWork() 方法进行重写
    protected override void OnDoWork (DoWorkEventArgs e)
    {
        ReportProgress (0, "对此报表执行操作...");
        //初始化财务报表数据
    }
}
```

```

while (!finished report )
{
    if (CancellationPending)
    {
        e.Cancel = true;
        return;
    }
    //进入另一个计算步骤
    ReportProgress (percentCompleteCalc, "另一个计算步骤...");
}
ReportProgress (100, "完成!");
e.Result = Result = completed report data;
}
}

```

使用上面的代码，无论是谁调用了 `GetFinancialTotalsBackground()` 方法都将得到一个 `FinancialWorker`——一个管理实际需要的后台操作的类。这个类可以报告进度，可以取消，并且不需要调用 `Control.Invoke()` 方法即可与 Windows 窗体同步。

 说明：在上例中，`FinancialWorker` 类同样具有异常处理，并且使用了标准协议。

22.11 用于读/写操作的锁

在多线程的程序中，需要对读/写操作进行保护。写操作是排他性的（即同一时刻只能由一个线程写入），而读操作可由多个线程进行读取。这时可以使用 `ReaderWriterLockSlim` 来保护多个线程读取但每次只采用一个线程写入的资源。

22.11.1 了解读/写操作的锁

通常情况下，一个线程安全的类的实例可以支持同步的读操作，但是并不支持同步的更新。这一点对于资源（例如文件）来说也同样成立。即使是拥有可执行的锁类的受保护的实例也是如此，如果对其进行很多的读操作和一个偶尔发生的更新操作时，该实例依旧会没有原因地禁止这种并发行为。举例来说，在通常的应用程序服务器中，经常在缓存里对数据进行操作。`ReaderWriterLockSlim` 类就是被设置为这种情况时，提供最大数量的可应用的锁。

`ReaderWriterLockSlim` 类和 `ReaderWriterLock` 类都可以提供两种锁：对于读操作的锁和对于写操作的锁。写操作锁通常是排外的，而读操作锁则是可以共存的。也就是说，如果一个线程拥有了写操作的锁，那么它会阻止其他的线程再拥有读或写的锁。但是如果没有线程拥有写操作的锁，那么这些线程都可以拥有读操作的锁。

关于读操作和写操作的锁的定义和释放可以通过以下代码完成：

```

public void EnterReadLock();
public void ExitReadLock();
public void EnterWriteLock();
public void ExitWriteLock();

```


另外, 对于 `EnterReadLock()` 方法和 `EnterWriteLock()` 方法都可以用 `try` 语句包围起来, 来接受 `Monitor.TryEnter` 形式的关于超时的参数。`ReaderWriterLock` 类还提供了与此相类似的方法, 命名为 `AcquireXXX` 和 `ReleaseXXX`。这两种方法将会在超时发生时, 抛出异常并返回 `false`。

22.11.2 管理资源访问锁定状态类 `ReaderWriterLockSlim`

下例中, 演示了 `ReaderWriterLockSlim` 类的具体用法。将会有 3 个线程持续地枚举一个列表, 另外两个线程每秒钟向列表添加一个随机数字。一个读操作的锁用来控制线程的读操作, 一个写操作的锁用来控制线程的写操作。

```
class SlimDemo
{
    static ReaderWriterLockSlim rw = new ReaderWriterLockSlim();
    static List<int> items = new List<int>();
    static Random rand = new Random();
    static void Main()
    {
        //在新线程上启动方法 Read()
        new Thread (Read).Start();
        //在新线程上启动方法 Read()
        new Thread (Read).Start();
        //在新线程上启动方法 Read()
        new Thread (Read).Start();
        //在新线程上启动方法 Write ()
        new Thread (Write).Start ("a");
        //在新线程上启动方法 Write ()
        new Thread (Write).Start ("b");
    }
    //静态方法 Read()
    static void Read()
    {
        //判断条件永远成立
        while (true)
        {
            rw.EnterReadLock();
            foreach (int i in items)
            {
                Thread.Sleep (10);
            }
            rw.ExitReadLock();
        }
    }
    //静态方法 Write ()
    static void Write (object threadID)
    {
        //判断条件永远成立
        while (true)
        {
            int newNumber = GetRandNum (100);
            rw.EnterWriteLock();
            items.Add (newNumber);
            rw.ExitWriteLock();
            Console.WriteLine ("线程 " + threadID + " 添加 " + newNumber);
        }
    }
}
```

```

        Thread.Sleep (100);
    }
}
//静态方法 GetRandNum ()
static int GetRandNum (int max)
{
    lock (rand) return rand.Next (max);
}
}

```

运行结果如下所示。

```

线程 b 添加 76
线程 a 添加 23
线程 b 添加 17
线程 a 添加 98
...

```

ReaderWriterLockSlim 类与简单的锁相比，允许更多的并发读操作。可以通过在 while 循环中的 Write()方法前添加如下的代码实现这一点：

```
Console.WriteLine (rw.CurrentReadCount + "个并行的读操作");
```

程序将会打印出“3 个并行的读操作”。


22.11.3 读/写操作锁的进一步说明

在一个原子操作中，有时候可以将一个读操作的锁换为一个写操作的锁。例如，假设程序希望在列表中不含有某一项时将此项加载到这个列表中，那么，在理想状态下，开发人员希望持有可执行锁的时间最短，就可以按以下步骤执行：

- (1) 获得一个读操作的锁。
- (2) 测试在当前列表中是否已经存在了此项，如果存在，则将锁释放并返回。
- (3) 释放读操作锁。
- (4) 获得一个写操作的锁。
- (5) 添加项。

这个解决方案的问题在于，可能会有另一个线程在第 (3) 和第 (4) 步间更改了列表（比如增加了相同的项）。ReaderWriterLockSlim 通过第 3 种锁（upgradeable lock）解决了这一问题。第 3 种升级的锁类似于一个读操作锁，不同的是它可以在某个原子操作中起到写操作锁的作用，下面是使用这个锁的主要步骤：

- (1) 调用 EnterUpgradeableReadLock。
- (2) 执行读操作（比如测试某项是否已存在于列表中）。
- (3) 调用 EnterWriteLock（这个方法可以将一个升级锁转换成一个写操作的锁）。
- (4) 执行写操作（比如向链表中添加项）。
- (5) 调用 ExitWriteLock（这个方法可以将一个写操作的锁转换成一个升级锁）。
- (6) 执行其他读操作。
- (7) 调用 ExitUpgradeableReadLock。

说明：一个升级的锁类似于一个嵌套锁或者递归锁。从功能上看，在执行步骤的第 3 步中，ReaderWriterLockSlim 类释放了一个读操作锁，并获得了一个写操作锁。

升级的锁和读操作的锁最大的不同就是，几个可升级的锁可以与任意多个读操作的锁共存，但是在同一时候，只能够存在一个可升级的锁。

下面的程序演示了使用一个可升级的锁向一个列表中添加项的过程，代码如下：

```
//使用一个可升级的锁向一个列表中添加项
//判断条件永远成立
while (true)
{
    //获得随机数
    int newNumber = GetRandNum (100);
    //获得可升级的锁
    rw.EnterUpgradeableReadLock();
    //如果列表中不包含新产生的随机数
    if (!items.Contains (newNumber))
    {
        //获得写操作锁
        rw.EnterWriteLock();
        //项列表中添加新项
        items.Add (newNumber);
        //释放写操作锁
        rw.ExitWriteLock();
        //打印提示信息
        Console.WriteLine ("线程" + threadID + " 添加" + newNumber);
    }
    //释放可升级的锁
    rw.ExitUpgradeableReadLock();
    //线程挂起
    Thread.Sleep (100);
}
```

22.12 用线程池管理线程

如果在应用程序中有很多的线程，并且这些线程的大部分时间都处于等待的阻塞状态，那么可以通过线程池来减少资源的负担。线程池通过将等待操作集中在几个线程上执行来减少资源的消耗。

使用线程池时，可以通过注册一个 Wait 句柄的代理来执行等待操作，这个方法可以通过调用 ThreadPool.RegisterWaitForSingleObject 方法来实现。如下面的代码所示。

```
class Test
{
    static ManualResetEvent starter = new ManualResetEvent (false);
    //主程序入口
    public static void Main()
    {
        //注册一个 Wait 句柄的代理来执行等待操作
        ThreadPool.RegisterWaitForSingleObject (starter, Go, "执行", -1, true);
        //线程挂起
    }
}
```

```

Thread.Sleep (5000);
Console.WriteLine ("信号量...");
starter.Set();
Console.ReadLine();
}
public static void Go (object data, bool timedOut)
{
    Console.WriteLine ("开始" + data);
    // 执行任务...
}
}

```


这段程序的运行结果如下所示。

```

信号量...
开始执行

```

所有的线程池线程都是后台线程，也就是说当应用程序的前台线程终止时，这些线程能够自动结束。然后，如果一个线程希望在应用程序退出前等待，直到线程中的重要任务运行完成，那么可以通过在线程上调用 `Join()` 方法来实现。

 **注意：**线程可以结束，而线程池却不会结束。

另外，也可以不使用等待句柄而是通过调用 `QueueUserWorkItem()` 方法来使用线程池。使用线程池有一点好处，就是线程池中的线程数目是一定的，当任务数量超过线程数量时，任务将会自动排队等待。在下面的例子中，一共有 100 个任务需要使用线程池，但是线程池一次只能执行 25 个任务。具体实现方法如下所示。

```

class Test
{
    static object workerLocker = new object ();
    static int runningWorkers = 100;
    //程序入口
    public static void Main()
    {
        //
        for (int i = 0; i < runningWorkers; i++)
        {
            //使用线程池
            ThreadPool.QueueUserWorkItem (Go, i);
        }
        //打印信息
        Console.WriteLine ("等待线程被完成...");
        //资源上锁
        lock (workerLocker)
        {
            //当运行的线程数大于 0 时
            while (runningWorkers > 0)
            {
                Monitor.Wait (workerLocker);
            }
        }
        Console.WriteLine ("Complete!");
        Console.ReadLine();
    }
}

```



```

}
//静态方法 Go()
public static void Go (object instance)
{
    Console.WriteLine ("开始: " + instance);
    Thread.Sleep (1000);
    Console.WriteLine ("结束: " + instance);
    lock (workerLocker)
    {
        runningWorkers--;
        Monitor.Pulse (workerLocker);
    }
}
}
}

```

为了将更多的对象传递给目标方法，可以定义一个具有全部属性的自定义对象，或者调用其他方法。例如，如果一个 Go()方法接收了两个整数参数，则可以使用下面的代码启动：

```
ThreadPool.QueueUserWorkItem (delegate (object notUsed) { Go (11,22); });
```

22.13 用异步代理得到线程返回的数据

在本章的开始部分，讲述了如何使用 `ParameterizedThreadStart` 将数据传递给一个线程。但是，有时还需要另一种实现机制使线程执行完毕后得到从线程中返回的数据。异步代理为这个需求提供了实现的机制，它可以允许任意数量和类型的参数传入。另外，在异步代理中如果有未捕获的异常出现，则会在原线程上被捕获，因此不需要投入太多的精力在异步代理的异常处理上。异步代理也提供了使用线程池的另一种方法。

在开始学习使用异步代理前，首先将讨论一个常见的、同步的程序模型。假定需要比较两个网页，那么可以依次下载这两个网页并比较它们的输出，具体代码如下所示。

```

static void ComparePages()
{
    WebClient wc = new WebClient ();
    string s1 = wc.DownloadString ("http://www.microsoft.com");
    string s2 = wc.DownloadString ("http://microsoft.com");
    Console.WriteLine (s1 == s2 ? "相同" : "不同");
}

```

在上面的程序中，如果能够实现同时下载两个网页则会使程序运行得更快。在下面的代码中，通过使用异步代理来下载这两个网页，并且比较得出结果。

```

delegate string DownloadString (string uri);
static void ComparePages()
{
    //实例化代理
    DownloadString download1 = new WebClient().DownloadString;
    DownloadString download2 = new WebClient().DownloadString;
    // 开始下载
    IAsyncResult cookie1 = download1.BeginInvoke (uri1, null, null);
    IAsyncResult cookie2 = download2.BeginInvoke (uri2, null, null);
}


```

```
//执行随机计算
double seed = 1.23;
for (int i = 0; i < 1000000; i++)
{
    seed = Math.Sqrt (seed + 1000);
}
//获得下载的结果, 如果需要的话等待执行完成
string s1 = download1.EndInvoke (cookie1);
string s2 = download2.EndInvoke (cookie2);
Console.WriteLine (s1 == s2 ? "相同" : "不同");
}
```

通过声明和实例化希望异步执行的方法来执行程序。在上面的例子中需要两个代理, 每个代理处理一个单独的网络客户端对象。首先将调用 `BeginInvoke()` 方法, 为了与代理相一致, 必须向 `BeginInvoke()` 方法传入一个字符串, 并在代理上标注 `BeginInvoke()` 方法和 `EndInvoke()` 方法的类型。

`BeginInvoke()` 方法还需要两个参数, 一个是在异步方法完成时所需要执行的方法, 另一个是用户自定义的数据类型。在通常情况下, 这两个参数是不需要的, 可以设为 `null`。`BeginInvoke()` 方法返回一个 `IASynchResult` 类型的对象。`IASynchResult` 类型同样有一个 `IsCompleted` 属性来检查进度。

其次, 将在代理上调用 `EndInvoke()` 方法。如果需要的话, `EndInvoke()` 方法将等到它的方法执行完毕, 然后返回一个在代理中指明的方法返回值。`EndInvoke()` 方法的好处在于, 如果 `DownloadString()` 方法有 `ref` 或 `out` 型的参数, 则将被加载到 `EndInvoke()` 的信号量中, 允许向调用者返回多个值。

 **注意:** 如果在异步方法的执行中遇到了没有处理的异常, 它将被传递给调用 `EndInvoke()` 方法的线程中。

22.14 .NET 提供的计时器

如果希望方法周期性的执行, 最简单的方法是使用计时器。计时器由 `System.Threading` 命名空间的 `Timer` 类给出。线程计时器比线程池技术在多个计时器的创建方面有了更大的优越性。`Timer` 类是一个相对简单的类, 常用的只有构造函数和两个方法。具体如下所示。

```
public sealed class Timer : MarshalByRefObject, IDisposable
{
    public Timer (TimerCallback tick, object state, int dueTime,
        int period);
    //改变时间间隔
    //1st 表示第一次运行的时间
    //后面的间隔时间
    public bool Change (int dueTime, int period);
    //关闭计时器
    public void Dispose();
}
```

在下例中, 将有一个计时器在 5 秒后调用 `Tick()` 方法, 每 1 秒钟打印出一个“滴答”:


```

using System;
using System.Threading;
class Program
{
    static void Main()
    {
        Timer tmr = new Timer (Tick, "滴答.", 5000, 1000);
        Console.ReadLine();
        //结束计时器
        tmr.Dispose();
    }
    static void Tick (object data)
    {
        //运行在线程池中
        //打印出一个“滴答”
        Console.WriteLine (data);
    }
}

```

.NET 框架还提供了另一个计时器类，在 `System.Timers` 命名空间中。与 `System.Threading` 命名空间的 `Timer` 类相比，具有以下新特性：

- ☐ 可以被当做组件使用，可以被拖曳到 Visual Studio 窗体设计器中；
- ☐ 拥有一个 `Interval` 属性；
- ☐ 拥有一个 `Elapsed` 事件代替了 callback 代理；
- ☐ 拥有一个 `Enable` 属性来开始或暂停计时器；
- ☐ 拥有 `Start()` 方法和 `Stop()` 方法。

在下面的例子中，将演示 `System.Timers.Timer` 类的用法。

```

using System;
//Timers 类的命名空间
using System.Timers;
class SystemTimer
{
    static void Main()
    {
        //构造函数，不需要传入参数
        Timer tmr = new Timer();
        tmr.Interval = 500;
        //使用事件代替代理
        tmr.Elapsed += tmr_Elapsed;
        //启动计时器
        tmr.Start();
        Console.ReadLine();
        //暂停计时器
        tmr.Stop();
        Console.ReadLine();
        //继续运行计时器
        tmr.Start();
        Console.ReadLine();
        //停止计时器
        tmr.Dispose();
    }
    static void tmr_Elapsed (object sender, EventArgs e)
    {
        Console.WriteLine ("滴答");
    }
}


```

```

    }
}

```

.NET 框架还提供了第 3 种计时器，位于 `System.Windows.Forms` 命名空间中，这个类已经在前面的章节中进行了介绍，在此处就不再赘述。

 **说明：**一个 Windows 窗体计时器并没有使用线程池技术，而是在创建计时器对象的线程上引发 Tick 事件，即使这个线程是应用程序的主线程。事实上 Windows 窗体计时器是一个单线程计时器。

22.15 各线程数据的局部存储

局部存储就是将每一个线程的数据与其他线程的数据相隔离，单独存储起来。对于局部存储的使用主要有以下两种常用方法：

- ❑ `Thread.GetData()` 方法主要用于读取线程中的独立的存储数据。
- ❑ `Thread.SetData()` 方法主要用于向线程中写入独立的存储数据。

在使用这两个方法时，都需要传入一个 `LocalDataStoreSlot` 对象。下面是使用这两个方法的具体例子：

```

class ...
{
    // 相同的 LocalDataStoreSlot 对象可以用于不同的线程中
    LocalDataStoreSlot secSlot = Thread.GetNamedDataSlot ("securityLevel");
    //SecurityLevel 对于不同的线程拥有独立的值
    int SecurityLevel
    {
        get
        {
            object data = Thread.GetData (secSlot);
            return data == null ? 0 : (int) data;
        }
        set
        {
            Thread.SetData (secSlot, value);
        }
    }
}
...

```

另外，还可以使用 `Thread.FreeNamedDataSlot()` 方法将在线程间释放给出的数据。

22.16 本章总结


在本章中，主要就 C# 的多线程编程进行了介绍。首先，本章通过几个简单的例子引入了什么是多线程编程、线程的工作原理和何时需要进行多线程编程。然后介绍了在多线程编程中，最简单也是最常用的类 `Thread`，并阐述了如何创建一个线程、如何操控一个线程，以及线程的状态与线程安全等。最后介绍了常见的多线程编程技术及与此相关的类。

22.17 实战练习


1. 在 Visual Studio 2010 中新建一个控制台应用程序，编写一个多线程类，该类的构造方法调用 `Thread` 类带字符串参数的构造方法。建立自己的线程名，然后随机生成一个休眠时间，再将自己的线程名和休眠多长时间显示出来。

2. 在 Visual Studio 2010 中新建一个控制台应用程序，编写一个用线程实现一个数字时钟的应用程序。该线程类要采用休眠的方式，把绝大部分时间让系统使用。

3. 在 Visual Studio 2010 中新建一个控制台应用程序，创建一个线程，指定一个限定时间（如 60s），线程运行时，大约每 3s 输出一次当前所剩时间，直至给定的限定时间用完。

 **提示：**通过 `sleep` 方法让线程休眠 3s。

4. 在 Visual Studio 2010 中新建一个控制台应用程序，用来模拟订票业务。创建一个名为 `BookingClerk` 类，代表自动售票员，其中包含一个订票方法 `booking`。假设一开始有 10 张票可预定，程序运行时产生两个订票客户同时自动向自动售票员订票。

 **提示：**通过 `lock`（或其他锁定资源）的方法来处理。

第 6 篇 Web 数据库开发

- ▶▶ 第 23 章 数据库基础知识
- ▶▶ 第 24 章 ADO.NET 数据库编程
- ▶▶ 第 25 章 ASP.NET 技术入门
- ▶▶ 第 26 章 服务器端控件详解

第 23 章 数据库基础知识

数据库的发展历史很悠久，从 19 世纪 60 年代开始出现的网状、层次数据库系统，到今天的以面向对象为特征的数据库系统，数据库已经经历了数次变革。数据库是利用计算机技术统一管理的相关数据的集合，它能动态地存储大量的相互关联的数据。Microsoft SQL Server 2008 是本章将要重点介绍的内容，它是微软推出的最新数据库，相比之前的 SQL 数据库系列，它在很多方面都有改进，如全文检索、查询引擎、统计信息等。因为本书的重点不在数据库本身，所以对这些细节就不再详述。

23.1 了解 SQL Server

SQL Server 2008 是一个全面的、集成的、端到端的数据解决方案，它为用户提供了一个安全、可靠和高效的平台用于企业数据管理和商业智能应用方面。SQL Server 2008 为数据处理和开发者带来强大而熟悉的管理软件，同时降低了数据系统的多平台上创建、部署、管理、使用和分析的复杂度。通过全面的功能集和现有系统的集成性，以及对日常任务的自动化管理能力，SQL Server 2008 为不同规模的企业提供了一个完整的数据解决方案。

SQL Server 2008 可以帮助用户随时随地管理任何数据。它可以将结构化、半结构化和非结构化文档的数据（例如图像和音乐）直接存储到数据库中。SQL Server 2008 提供一系列丰富的集成服务，可以对数据进行查询、搜索、同步、报告和分析之类的操作。数据可以存储在各种设备上，从数据中心最大的服务器一直到桌面计算机和移动设备，用户可以控制数据而不用管数据存储在哪里。

SQL Server 2008 允许用户在使用 Microsoft .NET 和 Visual Studio 开发的自定义应用程序中使用数据，信息工作人员也可以通过他们日常使用的工具（如 Office 2007/2010）直接从数据库中访问数据。

如图 23.1 所示为 SQL Server 2008 数据平台的组成架构。

SQL Server 2008 平台有以下特点。

- ☐ 可信的：使得公司可以以很高的安全性、可靠性和可扩展性来运行最关键任务的应用程序。
- ☐ 高效的：使得公司可以降低开发和管理数据基础设施的时间和成本。
- ☐ 智能的：提供了一个全面的平台，可以在用户需要的时候给他发送观察和信息。

从“可信的”角度来看，SQL Server 2008 在 SQL Server 2005 的基础上做了很多安全性方面的增强，下面简单列几项：

加密功能，可以对整个数据库、数据文件和日志文件进行加密，而不需要改动应用程序。进行加密使公司可以满足遵守规范及其关注数据隐私的要求。



图 23.1 SQL Server 2008 数据平台

增强了审查，SQL Server 2008 让用户可以审查数据的操作，从而提高了遵从性和安全性。审查不仅包括对数据修改的所有信息，还包括关于什么时候对数据进行读取的信息。

当然，从可信任角度来看，SQL Server 2008 还增加了确保业务可持续性的一些功能。

从“高效的”角度来看，SQL Server 2008 从降低管理系统的时间和成本，使用 ADO.NET、LINQ 等技术加速开发过程，改进 Transact-SQL 体验等方面提供了很多全新的功能。

从“智能的”角度来看，SQL Server 2008 提供了一个全面的平台，用于当用户需要时可以为它提供智能化。如能集成任何数据、数据压缩、备份压缩、分区表并行、新的报表设计器、预测分析等功能都能为用户提供智能化。

SQL Server 2008 是一个重大的产品版本，它推出了许多新的特性和关键的改进，使得它成为迄今为止最强大和最全面的 SQL Server 版本。

SQL Server 2008 是一个庞大的数据库管理系统，包含的内容非常多，本书只是介绍它最基本的功能——数据库管理，同时介绍它与 Visual Studio 2010 的集成。关于 SQL Server 2008 更多功能的使用，读者可以参见联机帮助或是相关网站和书籍。

通常，在安装 Visual Studio 2010 时可选择安装免费版 SQL Server 2008 Express。

23.2 操作 MSSQL 数据表

表是数据库中的主要对象，用于存储各种信息，它是数据库中其他对象的基础。数据库中的表一般分为永久性表和临时表，本节只讨论永久性表。对于表的操作包括创建、修

改和删除表的结构,以及查询表中的数据等,这些操作都是使用 SQL 语句来完成,本节将进行详细说明。

23.2.1 在数据库中创建、修改和删除表

表的创建需要用到 CREATE 关键字,下面通过实例进行说明。SQL 语句代码如下:

```
CREATE DATABASE mydb
CREATE TABLE student(
UID varchar(20) not null,
Name varchar(10) not null,
Sex varchar(10) not null,
Age varchar(5) not null,
Tel varchar(20) not null,
Address varchar(50) null)
```

在 mydb 数据库里创建了名为 student 的表,该表具有 6 列,每一列由列名、数据类型和是否为空属性组成。其中,UID 表示学生的学号,数据类型为 varchar,长度为 20,不能为空;Name 表示学生的姓名,数据类型为 varchar,长度为 10,不能为空;Sex 表示学生的性别,数据类型为 varchar,长度为 10,不能为空;Age 表示学生的年龄,数据类型为 varchar,长度为 5,不能为空;Tel 表示学生的电话,数据类型为 varchar,长度为 20,不能为空;Address 表示学生的地址,数据类型为 varchar,长度为 50,可以为空。

如果用户在创建表的过程中忽略了某些因素,则需要对表的结构进行修改。如果需要在当前表中添加一列,可使用如下 SQL 语句:

```
ALTER TABLE student ADD birthday varchar(20) null
```

为 student 表添加了一个新列 birthday,数据类型为 varchar,长度为 20,可以为空。如果需要删除该列,可使用如下 SQL 代码:

```
ALTER TABLE student DROP birthday
```

表的删除语句比较简单,SQL 代码如下所示。

```
DROP table_name
```

23.2.2 向表中插入、修改、删除和检索数据

创建好表的结构之后,大多数时间都是使用 SQL 代码操纵表中的数据,操纵数据主要包括插入、修改、删除数据等主要操作,下面将详细介绍。

(1) 插入数据:表是用来存储数据的,所以在表创建完以后需要向表中插入数据。常用的操作是使用 INSERT INTO...VALUES 语句,如下所示。

```
INSERT INTO student
VALUES('2001022117','huanglei','male','19','123456','Chengdu','1986-5-8')
INSERT INTO student
VALUES('2001022118','liufang','female','20','123457','Chengdu','1986-9-8')
INSERT INTO student
```



```
VALUES ('2001022119', 'wanghong', 'female', '21', '123458', 'Beijin', '1986-9-23')
INSERT INTO student
VALUES ('2001022116', 'yangyang', 'female', '22', '123459', 'Dalian', '1986-9-12')
```

以上代码为表 student 插入 4 行数据，需要注意的是，插入的数据应该用单引号包括起来。执行完毕，打开表可看到如图 23.2 所示的 4 行数据。

	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022117	huanglei	male	19	123456	Chengdu	1986-5-8
2	2001022118	liufang	female	20	123457	Chengdu	1986-9-8
3	2001022119	wanghong	female	21	123458	Beijin	1986-9-23
4	2001022116	yangyang	female	22	123459	Dalian	1986-9-12

图 23.2 为表 student 添加数据

(2) 修改数据：UPDATE 语句可以修改表中的一行或多行数据，它的用法如下所示。

```
UPDATE table_name
SET column_name=expression
WHERE search_condition
```

WHERE 子句用于指定需要修改的行，SET 子句用于生成新的数据。例如，现在需要将表 student 的 Age 列全部加一，代码如下所示。

```
UPDATE student
SET Age=Age+1
```

修改后的结果如图 23.3 所示。

	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022117	huanglei	male	20	123456	Chengdu	1986-5-8
2	2001022118	liufang	female	21	123457	Chengdu	1986-9-8
3	2001022119	wanghong	female	22	123458	Beijin	1986-9-23
4	2001022116	yangyang	female	23	123459	Dalian	1986-9-12

图 23.3 修改表 student 后的结果

如果需要对表 student 中的某个学生的电话进行修改，其代码如下所示。

```
UPDATE student
SET Tel='654321'
WHERE UID='2001022117'
```

修改后数据库表的内容如图 23.4 所示。

	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022117	huanglei	male	20	654321	Chengdu	1986-5-8
2	2001022118	liufang	female	21	123457	Chengdu	1986-9-8
3	2001022119	wanghong	female	22	123458	Beijin	1986-9-23
4	2001022116	yangyang	female	23	123459	Dalian	1986-9-12

图 23.4 修改表 student 后的结果

(3) 删除数据: DELETE 语句用于删除表中的一行或多行数据, 它的用法如下所示。

```
DELETE FROM table_name
WHERE search_condition
```

WHERE 子句用于指定需要修改的行。比如需要删除年龄大于等于 21 的学生信息, 代码如下所示。

```
DELETE FROM student
WHERE Age>='21'
```

(4) 检索数据: 表中的数据检索是用户的常见操作, 通过检索数据可以把数据库中满足用户需求的信息提取出来。数据检索需要用到 SELECT 语句, 它允许从表中检索一个或多个行或列, 其用法可以很简单, 也可以很复杂, 下面通过例子进行说明。

查询表 student 中的所有内容, 代码如下:

```
SELECT * FROM student
```

运行结果如图 23.5 所示。

结果		消息					
	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022117	huanglei	male	20	654321	Chengdu	1986-5-8
2	2001022118	liufang	female	21	123457	Chengdu	1986-9-8
3	2001022119	wanghong	female	22	123458	Beijin	1986-9-23
4	2001022116	yangyang	female	23	123459	Dalian	1986-9-12

图 23.5 查询后的结果

改变列标题, 代码如下:

```
SELECT 'The birthday of'='The birthday of',Name,'IS'='is',birthday FROM student
```

“=” 可用于改变列标题, 执行结果如图 23.6 所示。

结果		消息			
	The birthday of	Name	IS	birthday	
1	The birthday of	huanglei	is	1986-5-8	
2	The birthday of	liufang	is	1986-9-8	
3	The birthday of	wanghong	is	1986-9-23	
4	The birthday of	yangyang	is	1986-9-12	

图 23.6 查询后的结果

查询行, 代码如下:

```
SELECT * FROM student WHERE Age>'20'
```

表示查询表 student 中年龄大于 20 的学生信息, 执行结果如图 23.7 所示。

```
SELECT * FROM student WHERE Name LIKE 'yang%'
```

表示查询表 student 中名字以 yang 开头的学生信息, 执行结果如图 23.8 所示。

	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022118	lufang	female	21	123457	Chengdu	1986-9-8
2	2001022119	wanghong	female	22	123458	Beijn	1986-9-23
3	2001022116	yangyang	female	23	123459	Dalian	1986-9-12

图 23.7 查询后的结果

	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022116	yangyang	female	23	123459	Dalian	1986-9-12

图 23.8 查询后的结果

该查询也称为匹配查询，在实际应用中会经常遇到，比如图书查询系统。LIKE 关键字具有 4 种匹配符，如表 23.1 所示。

表 23.1 LIKE 关键字的匹配符

匹 配 符	说 明	匹 配 符	说 明
%	表示0个或多个任意字符	[]	表示在指定范围内的任意单个字符
_(下画线)	表示任意单个字符	[^]	不在任意范围内的任意单个字符

说明如下所示。

LIKE '%Cc%'	返回包含“Cc”的任意字符串
LIKE '_Cc'	返回以“Cc”结尾的长度为 3 的字符串
LIKE '[Cc%]'	返回以“c”或“C”开头的任意字符串
LIKE '[^C]%'	返回首字符不为“c”的任意字符串

以上大致介绍了几种基本的数据检索，因为本书并非完全针对数据库，所以有关高级检索的内容这里不再详述。

23.2.3 设置表的主键约束

本节将介绍约束机制在表的管理中的应用，约束能确保数据完整性，通过限制表中行或列中的数据，以及表之间的数据来实现数据完整性。约束有几种类型，分别为主键约束、外键约束、唯一性约束、CHECK 约束和默认约束。下面将逐一进行介绍。

主键约束主要用于表的某一列或多列中都只具有唯一的值，这样能确保数据的唯一性。主键约束所包含的列不能具有空值，其语法形式如下所示。

```
CONSTRAINT constraint_name
    PRIMARY KEY [CLUSTERED|NONCLUSTERED] (column[,...n])
```

其中，CONSTRAINT 是约束的关键字，constraint_name 是指约束的名称，如果默认，系统会自动生成。PRIMARY KEY 表示主键，CLUSTERED 表示该主键约束是唯一性聚簇索引，NONCLUSTERED 表示该主键约束是唯一性非聚簇索引，默认情况下是 CLUSTERED。

下面仍然以表 student 为例进行说明，代码如下所示。

```
CREATE TABLE student(
    UID varchar(20) not null,
    Name varchar(10) not null,
    Sex varchar(10) not null,
    Age varchar(5) not null,
    Tel varchar(20) not null,
    Address varchar(50) null,
    CONSTRAINT stu_id PRIMARY KEY (UID)
)
```

以上代码在创建表 `student` 时，为 `UID` 列添加主键约束，这样可以保证每个学生的学号都是唯一的。也可以在创建完表以后再添加主键约束，代码如下所示。

```
ALTER TABLE student
ADD CONSTRAINT stu_uid PRIMARY KEY(UID)
```

23.2.4 设置表的外键约束

外键约束用于限制一个表中的某些列与其他表中的某些列的关联，从而实现表之间的依赖关系。它的语法格式如下所示。

```
CONSTRAINT constraint_name
FOREIGN KEY [CLUSTERED|NONCLUSTERED] (column[,...n])
REFERENCES ref_table(ref_col[,...n])
```

其中，`FOREIGN KEY` 表示外键，`REFERENCES` 后面的表名是指需要关联的表名和表的列名。为了便于举例，下面再创建一个表 `mark`，用于存储学生的考试成绩，代码如下所示。

```
CREATE TABLE mark(
UID varchar(20) not null,
MID varchar(20) not null,
English varchar(5) not null,
Maths varchar(5) not null,
PE varchar(5) not null,
Biology varchar(5) not null,
Geography varchar(5) not null,
CONSTRAINT stu_id PRIMARY KEY(UID)
)
```

并插入以下数据：

```
INSERT INTO mark
VALUES('2001022116','2116','85','80','83','75','78')
INSERT INTO mark
VALUES('2001022117','2117','84','86','80','77','73')
INSERT INTO mark
VALUES('2001022118','2118','83','70','80','76','78')
INSERT INTO mark
VALUES('2001022119','2119','82','80','82','75','88')
```

以上代码创建了一个新表 `mark`，包括学生学号、准考证号和各科成绩，并为学号列添加主键约束。接下来创建外键约束，将表 `student` 和表 `mark` 关联起来，代码如下所示。

```
ALTER TABLE student
ADD CONSTRAINT fk_stu
FOREIGN KEY(UID)
REFERENCES mark(UID)
```

如果试图删除 `mark` 表中的数据，此时，由于约束的存在，只有在表 `student` 中没有依赖信息的才能被删除，否则将不能删除（即在 `mark` 表中只能删除 `student` 表中不存在的 `UID`）。

23.2.5 设置表的唯一性约束

这种约束用于限制表中的某一列或多列中不能存在相同的行数据，它和主键约束比较

类似。但需要注意它们之间的区别，一是在一个表中可以同时添加多个唯一性约束，而主键约束只能出现一次；二是在唯一性约束中，最多可以允许出现一个空值，而在主键约束中不允许出现空值。唯一性约束语法形式如下所示。

```
CONSTRAINT constraint_name
    UNIQUE [CLUSTERED|NONCLUSTERED] (column[,...n])
```

下面的例子是为表 mark 中的准考证号 MID 列添加唯一性约束，代码如下所示。

```
ALTER TABLE mark
ADD CONSTRAINT stu_mid UNIQUE (UID)
```

23.2.6 设置表的 CHECK 约束

CHECK 约束用于对表中的某一列数据进行范围限制，它的语法形式如下所示。

```
CONSTRAINT constraint_name
    CHECK (logic_expression)
```

例如，如果需要对表 student 中的 Sex 列添加检查约束，只允许输入 male 或 female 时，代码如下所示。

```
ALTER TABLE student
ADD CONSTRAINT stu_chk CHECK (Sex='female' OR Sex='male')
```

23.2.7 设置列的默认约束

如果向某个表插入数据时，数据的某一列中具有很多相同的元素，此时可以采用默认约束，定义它为需要输入的元素。其语法形式如下所示。

```
CONSTRAINT constraint_name
    DEFAULT constant_expression
```

比如，为表 student 的 Age 列添加默认约束，当年龄为 19 时，可以不用输入，代码如下所示。

```
ALTER TABLE student
ADD CONSTRAINT stu_age DEFAULT '19' FOR Age
```

23.3 数据库的存储过程

存储过程封装了一组可以重用的 Transact-SQL 语句。它存储在服务器上，支持用户自定义的变量（包括接受和返回用户参数），并且已经通过了预编译。存储过程类似于其他编程语言中使用的函数，可以对大量复杂的操作进行封装，只需要提供一个输入和输出接口供用户调用即可。

在 SQL 中运用存储过程是件很轻松快乐的事情，它具有以下优点。

- ❑ 存储过程只是在创建时才会编译，以后每次调用执行时均不再需要编译，这样可以大大提高数据库的访问速度。

- ❑ 存储过程可以封装大量复杂的 Transact-SQL 语句，让应用程序的可读性更强。
- ❑ 存储过程可以重复使用，有利于提高开发效率。
- ❑ 安全性较高，可以设置用户的访问权限。

存储过程可以分为几种类型，分别为系统存储过程、本地存储过程、临时存储过程、远程存储过程和扩展存储过程。本节只介绍本地存储过程，它是指创建在用户自己的数据库中的存储过程。

23.3.1 创建存储过程的 SQL 语句

存储过程用 PROCEDURE 语句表示，创建存储过程的基本语法格式如下所示。

```
CREATE PROCEDURE procedure_name
[ { @parameter data_type }
  [ VARYING ] [ = default ] [ OUTPUT ]
] [ ,...n ]
[ WITH
{ RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION } ]
[ FOR REPLICATION ]
AS
sql_statement [ ...n ]
```

procedure_name 是指存储过程的名称，parameter 是指存储过程中的参数，data_type 表示参数类型，VARYING 指定输出参数的结果集，OUTPUT 表示参数为返回参数，RECOMPILE 表示存储过程在运行时重新编译，ENCRYPTION 表示对存储过程进行加密，REPLICATION 指定不能在订阅服务器上执行为复制创建的存储过程，AS 表示存储过程要执行的操作。下面通过两段存储过程示例进行说明。存储过程一的内容如下：

```
CREATE PROCEDURE proc
@input varchar(5),
@output varchar(20) OUTPUT
AS
SELECT @output=@input+1;
```

这段存储过程创建了一个名为 proc 的存储过程，定义了两个参数，一个是输入参数 input，另一个是输出参数 output，且它是输入参数值加 1。

存储过程二的内容如下：

```
CREATE PROCEDURE stu_info
@partname varchar(20)
AS
SELECT Name,English
FROM student t INNER JOIN mark m
ON t.UID=m.UID
WHERE Name LIKE @partname
```

这段存储过程创建名为 stu_info 的存储过程，用于查询学生的姓名和英语成绩。partname 是用于匹配查询的输入参数，INNER JOIN 表示表之间的连接。如果是多个表之间的连接，则采用如下形式：

```
FROM table1 t1 INNER JOIN table2 t2
ON t1.columnname=t2.columnname INNER JOIN table3 t3
ON t2.columnname=t3.columnname INNER JOIN table4 t4
```



```
ON t2.columnname=t3.columnname
.....
```

以上代码通过两个简单的例子说明了存储过程的创建。如果用户对存储过程进行修改,则需要使用 ALTER PROCEDURE 语句,除了 ALTER 关键字之外,它的语法格式和创建存储过程是完全一样的,此处不再详述。

23.3.2 执行和删除存储过程的 SQL 语句

存储过程的执行可以直接运用 EXECUTE 命令,对于 23.3.1 节中的存储过程一,其执行代码如下所示。

```
DECLARE @input varchar(5)
SET @input='19'
DECLARE @output varchar(20)
EXECUTE stu_select @input,@output OUTPUT
PRINT @output
```

因为该存储过程带有输出参数,所以在创建和执行存储过程时均需要声明,并且需要指定 OUTPUT 关键字。

存储过程二因为不涉及输出参数,所以它的执行方式比较简单,如下所示。

```
EXECUTE stu_info @partname='huang%'
```

执行结果如图 23.9 所示。



	Name	English
1	huanglei	84

图 23.9 存储过程二执行查询结果

如果需要删除存储过程,则使用 DROP 语句,如下所示。

```
DROP PROCEDURE proc_name
```

23.3.3 用 SQL Server Management Studio 管理存储过程

在 SQL Server Management Studio 中,提供了存储过程向导,包括创建、修改和删除存储过程。本节将利用该向导,重新创建 23.3.1 节中的存储过程二,具体步骤如下所示。

安装 SQL Server 2008 Express 将不会安装 SQL Server Management Studio,读者可到微软网站免费下载安装。

(1) 打开 SQL Server Management Studio,转到 mydb 数据库目录下,选择“可编程性”|“存储过程”选项,右键单击该选项,选择“新建存储过程”命令,如图 23.10 所示。这是存储过程的模板,接下来需要对其进行编辑。

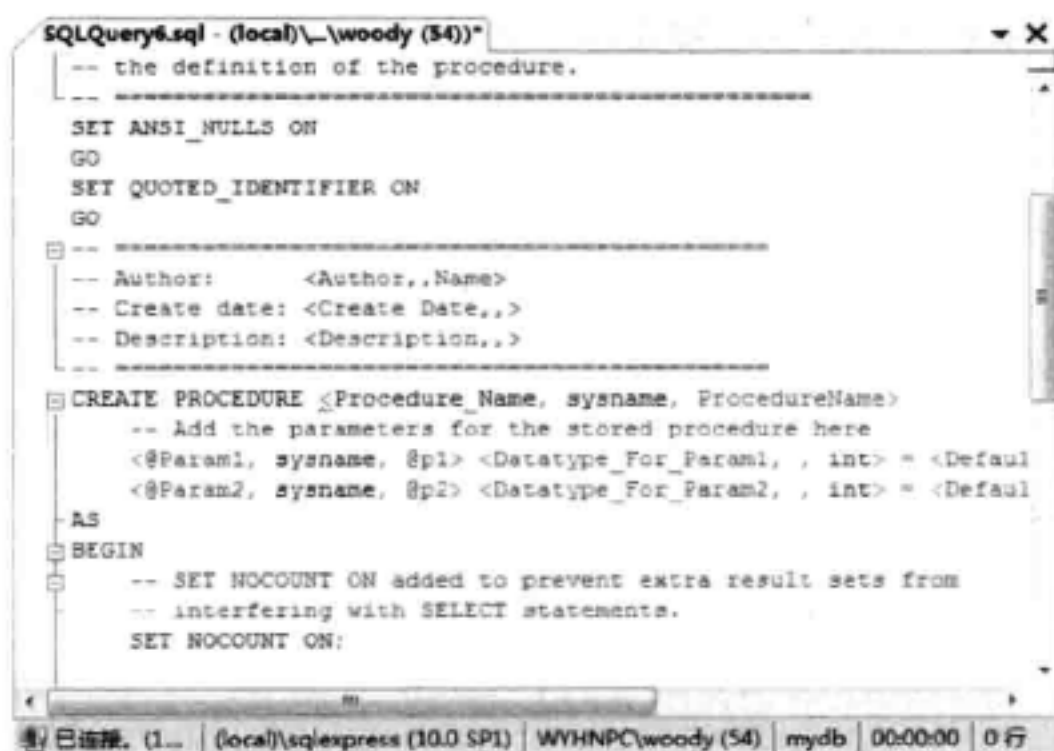


图 23.10 存储过程模板

(2) 选择“查询”|“指定模板参数的值”命令，输入需要的值，如图 23.11 所示。



图 23.11 设置存储过程参数

图 23.11 中包括创建存储过程的作者、创建日期、用途、输入参数名、参数类型和参数初始值等信息。此外，因为本例只有一个输入参数，所以第二个输入参数为空。

(3) 单击“确定”按钮，用以下语句替换模板中的 SELECT 语句：

```

SELECT t.Name,m.English
FROM student t INNER JOIN mark m
ON t.UID=m.UID
WHERE Name LIKE @partname

```

(4) 选择“查询”|“执行”命令，即可完成存储过程的创建。

(5) 如果需要执行该存储过程，可以选择“新建查询”命令，输入以下语句：

```
EXECUTE stu_info1 @partname='yang%'
```

查询结果如图 23.12 所示。

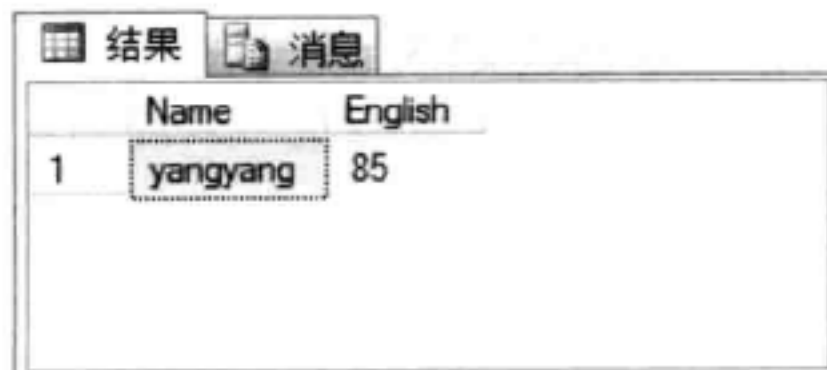


图 23.12 查询结果

如果需要修改存储过程，则转到存储过程所在目录（同步骤（1）类似），右键单击需要修改的存储过程，选择“修改”命令，修改完毕后，转到步骤（4）即可。

如果需要删除存储过程，右键单击需要删除的存储过程，在弹出的快捷菜单中单击“删除”按钮即可。

23.4 数据库中的触发器

触发器是一种特殊的存储过程，但触发器能自动执行。例如，试图对表进行 DELETE、INSERT 和 UPDATE 操作时，那么该表中相应操作类型的触发器就会被自动执行，以拒绝不合法的操作，从而达到保护数据的目的。触发器具有以下优点。

- 触发器是自动执行的，它被激活后，能执行其他相关操作。
- 触发器能对数据库中的相关表进行层叠更改，如在某表中创建触发器，当需要删除该表中的一列时，则可以使用触发器删除在其他表中与该列相关的内容。
- 触发器具有约束功能。触发器由 3 部分组成，包括触发语句、触发器限制和触发体。其中，触发语句是导致触发器执行的 Transact-SQL 语句，它可以是 INSERT、UPDATE 和 DELETE 等语句；触发器限制用于指定一个布尔表达式，如果为真，则执行触发体，否则不执行；触发体是指触发器限制为真时，需要执行的 Transact-SQL 语句。

23.4.1 创建和使用触发器的 SQL 语句

创建触发器的语法形式如下所示。

```
CREATE TRIGGER trigger_name
ON {table|view}
WITH ENCRYPTION
{FOR|AFTER|INSTEAD OF} {[DELETE] [,] [INSERT] [,] [UPDATE]}
AS
Sql_statement
```

其中，TRIGGER 是触发器的关键字，触发器可以在表或视图中创建。为了隐藏触发器创建文本，可以使用 WITH ENCRYPTION。AFTER 触发器是在数据变动（INSERT、UPDATE 和 DELETE 等操作）完成之后才被激活。而 INSTEAD OF 触发器是在数据变动完成之前被激活，并取代 INSERT、UPDATE 和 DELETE 等操作。

例如，若需要删除表 student 中某个学生的信息，同时需要删除它在表 mark 中的信息，则可创建如下所示的触发器。

```
CREATE TRIGGER stu_del
ON student
FOR DELETE
AS
DELETE FROM mark WHERE mark.UID=(SELECT UID FROM DELETED)
```

为了测试正确与否，下面从表 student 中删除一行数据，如下所示。

```
DELETE FROM student WHERE Name='huanglei'
```

执行完毕后, 分别对两个表进行查询, 结果如图 23.13 和图 23.14 所示。

	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022116	yangyang	female	23	123459	Dakan	1986-9-12
2	2001022118	lufang	female	21	123457	Chengdu	1986-9-8
3	2001022119	wanghong	female	22	123458	Beijin	1986-9-23

图 23.13 删除后的结果

	UID	MID	English	Maths	PE	Biology	Geography
1	2001022116	2116	85	80	83	75	78
2	2001022118	2118	83	70	80	76	78
3	2001022119	2119	82	80	82	75	88

图 23.14 触发器执行后的结果

触发器中可以包含许多 Transact-SQL 语句, 但有些语句是被禁止使用的, 如所有的 CREATE 语句、所有的 DROP 语句、ALTER TABLE 和 ALTER DATABASE 语句, 以及 GRANT 和 REVOKE 语句等。读者在使用的时候一定要注意。

触发器的删除有两种方式, 一是在触发器相关的表被删除时, 它会自动删除; 二是运用 DROP 语句进行删除, 如下所示。

```
DROP TRIGGER trigger_name
```

23.4.2 用 SQL Server Management Studio 管理触发器

同存储过程一样, SQL Server Management Studio 也提供了触发器向导, 本节将针对 23.4.1 节中的例子, 利用触发器向导进行实现, 步骤如下所示。

(1) 打开 SQL Server Management Studio, 转到表 student 所在的目录, 展开该表后, 右键单击“触发器”项, 选择“新建触发器”选项, 如图 23.15 所示。

(2) 选择“查询”|“指定模板参数的值”命令, 进行如图 23.16 所示的设置。其中包括触发器名、需要添加触发器的表名和触发器的类型等信息。



图 23.15 触发器模板



图 23.16 设置触发器参数

(3) 单击“确定”按钮, 在 END 前添加触发器语句, 如下所示。

```
DELETE FROM mark WHERE mark.UID=(SELECT UID FROM DELETED)
```

(4) 选择“查询”|“执行”命令, 如果执行无误, 则触发器创建成功。

(5) 下面测试该触发器的功能, 选择“新建查询”命令, 删除表 student 中的某行数据, 如下所示。


```
DELETE FROM student WHERE Name='yangyang'
```

此时，分别执行对两个表的全体查询，代码如下所示。

```
SELECT *FROM student
SELECT *FROM mark
```

执行结果如图 23.17 所示。

结果		消息					
	UID	Name	Sex	Age	Tel	Address	birthday
1	2001022118	liufang	female	21	123457	Chengdu	1986-9-8
2	2001022119	wanghong	female	22	123458	Beijin	1986-9-23

	UID	MID	English	Maths	PE	Biology	Geography
1	2001022118	2118	83	70	80	76	78
2	2001022119	2119	82	80	82	75	88

图 23.17 触发器执行结果

在 SQL Server Management Studio 中对触发器的修改和删除过程与存储过程相似，这里不再赘述。

23.5 本章总结

本章主要介绍了 SQL Server 2008 的相关基础知识，这是为 24 章将要介绍的 ADO.NET 访问数据库做准备。通过对表、存储过程、触发器的讲解，让读者能更好地理解数据库的结构及功能。Transact-SQL 语言是 SQL Server 2008 中的查询语言，它的功能强大并且简单易学，在应用程序中访问数据库会常常用到这些语言，读者应该好好掌握。

23.6 实战练习

1. 在 SQL Server Management Studio 中创建一个员工数据库 Employee，并创建两张表，其中一张表为 EmpInfo，用来保存员工的基本信息，包括工号、姓名、性别、年龄、职称、联系电话、家庭住址、工作部门等；另一张表为 Salary，用来保存员工每月的工资，包括工号、工资月份、基本工资、奖金、扣款等。

2. 修改 EmpInfo 表，将表 EmpInfo 中的工号作为主键；修改 Salary 表，为 Salary 表的工号创建外键，外键列为 EmpInfo 表中的工号。

3. 向表 EmpInfo 和 Salary 中添加测试数据，要求每个表中至少要有 5 条记录，且 Salary 表中的工号应在 EmpInfo 中出现。

4. 在 EmpInfo 表中创建一个触发器，当删除某位员工信息时，同时删除 Salary 表中的工资数据。

第 24 章 ADO.NET 数据库编程

随着计算机应用和软件技术的发展，软件需要处理的应用数据量以及数据之间的复杂程度不断加剧，因此对数据库的要求也日趋复杂。另外，各大数据库软件开发商不断推出自己的管理软件，也使得数据库的统一访问变得日趋重要。本章将介绍.NET 下统一的数据库访问组件——ADO.NET。

24.1 ADO.NET 介绍

ADO.NET 是活动数据对象（Active Data Objects，ADO）的更新版本，它主要用于访问数据。它支持 3 种数据访问格式，包括连接的数据访问、断开的数据库访问和 XML 访问。ADO.NET 解决了应用程序和数据库之间不能分离的问题，它让程序员能轻松地连接到数据库，并在数据库中进行检索、添加、删除、修改和插入等操作。它主要由两个部分构成，即 DataSet 和 .NET 数据提供者，其中 DataSet 用于以表格的形式存储数据；而 .NET 数据提供者能允许程序员与单独的数据源通信，常用的数据提供者包括 ODBC、SQL Server 和 OLE DB。

24.1.1 ADO.NET 是神马

ADO.NET 并不是 ADO 为了适应新的 .NET 框架而改进得到的版本，它是微软进行全新设计的新产品。与 ADO 的早期版本和其他数据访问组件相比，ADO.NET 具有以下优点。

- ❑ 互操作性：ADO.NET 应用程序可以利用 XML 的灵活性和广泛接受性。由于 XML 是用于在网络中传输数据集的格式，因此可以读取 XML 格式的任何组件都可以处理数据。实际上，接收组件根本不必是 ADO.NET 组件，传输组件可以只是将数据集传输给其目标，而不考虑接收组件的实现方式。目标组件可以是 Visual Studio 应用程序或无论用什么工具实现的其他任何应用程序，唯一的要求是接收组件能够读取 XML。
- ❑ 可编程性：.NET 类库中的 ADO.NET 数据组件以不同方式封装数据访问功能，帮助开发人员加快编程速度并减少犯错几率。已声明类型的数据集的代码更安全，原因在于它提供对类型的编译时检查。
- ❑ 性能：对于无连接的应用程序，ADO.NET 数据库提供的性能优于 ADO 无连接的记录集。当使用 COM 封送层在层间传输不连接的记录集时，会因将记录集内的值转换为 COM 可识别的数据类型而导致显著的处理开销。在 ADO.NET 中，这种数据类型转换则没有必要。

- 可伸缩性：因为 Web 可以极大增加对数据的需求，所以可缩放性变得很关键。Internet 应用程序具有无限的潜在用户供应，尽管应用程序可以很好地为十几个用户服务，但它不能保证向成百上千个（或几百万个）用户提供同样好的服务。ADO.NET 通过鼓励程序员节省有限资源来实现可缩放性，由于所有 ADO.NET 应用程序都使用对数据的不连接访问，因此它不会在较长持续时间内保留数据库锁或活动数据库连接。

ADO.NET 的优势是显而易见的，本章将给读者讲解如何通过 ADO.NET 轻松地访问数据库数据。

24.1.2 ADO.NET 相关的类和接口

System.Data 命名空间提供对表示 ADO.NET 结构的类的访问。通过 ADO.NET 可以生成一些组件，用于有效管理多个数据源的数据。在断开连接的情形中（如 Internet），ADO.NET 提供在多层系统中请求、更新和协调数据的工具。ADO.NET 结构也在客户端应用程序（如 Windows 窗体或 HTML 页）中实现。

ADO.NET 结构的中心构件是 DataSet 类，可以把它看成是一个临时的数据库对象。一个 DataSet 可以包含多个 DataTable 对象，每个 DataTable 都包含来自单个数据源的数据，例如，SQL Server、Access 等，可以把 DataTable 看成是一个临时的数据表。

每个 DataTable 包含一个 DataColumnCollection（DataColumn 对象的集合），DataColumnCollection 决定每个 DataTable 的架构，字段的顺序、类型、名称等属性。DataColumn 的 DataType 属性确定它所包含的数据的类型，而 ReadOnly 和 AllowDBNull 属性则进一步显示数据的有效性。

另外，DataSet 还通过 DataRelationCollection 来生成该数据库中数据表之间的关系集合。在添加一个 DataRelation 关系到 DataSet 时，将自动为相关的 Table 创建 UniqueConstraint 和 ForeignKeyConstraint。其中，UniqueConstraint 确保列中包含的值是唯一的，ForeignKeyConstraint 确定当主键值被更改或删除时将对子行或子列执行的操作。

System.Data 命名空间作为 ADO.NET 的根命名空间，提供了和 ADO.NET 相关的基本类和接口，表 24.1 列出了其中最常用的一些类。主要分成 DataSet、DataTable、DataView、DataRow、DataColumn、DataRelation 几个部分，它们各自还附带一些相关类，这里就不再罗列出来。

表 24.1 System.Data 常用类

类 名	说 明
DataSet	表示数据在内存中的缓存
DataTable	表示内存中数据的一个表
DataTableCollection	表示DataSet的表的集合
DataTableReader	DataTableReader以一个或多个只读、只进结果集的形式获取一个或多个DataTable对象的内容
DataRow	表示DataTable中的一行数据，即一个数据记录
DataRowCollection	表示DataTable的行的集合

续表

类 名	说 明
DataRowView	表示DataRow的自定义视图
DataColumn	表示DataTable中列的架构
DataColumnCollection	表示DataTable的DataColumn对象的集合
DataRowView	表示用于排序、筛选、搜索、编辑和导航的DataTable的可绑定数据的自定义视图
DataRelation	表示两个DataTable对象之间的父/子关系
DataRelationCollection	表示此DataSet的DataRelation对象的集合
Constraint	表示可在一个或多个DataColumn对象上强制的约束
ConstraintCollection	表示DataTable的约束的集合
DataException	表示使用ADO.NET组件发生错误时引发的异常

另外, System.Data 命名空间下还包括以下 4 个主要的 .NET Framework 数据提供程序, 将数据源的数据以 DataSet 的形式读取到内存。它们都有相应的 DataAdapter, 用做数据源和 DataSet 之间的桥梁, 从而实现不同数据源数据库的访问。

- ❑ System.Data.SqlClient 命名空间: 提供用于在 .NET 环境下访问 SQL Server 系列数据库的 .NET Framework 数据提供程序, 比如 SQL Server 2000 和 SQL Server 2008。
- ❑ System.Data.Odbc 命名空间: 提供用于在 .NET 环境下访问 ODBC 数据源的 .NET Framework 数据提供程序。
- ❑ System.Data.OleDb 命名空间: 提供用于在 .NET 环境下访问 Ole DB 数据源的 .NET Framework 数据提供程序, 比如 Access。
- ❑ System.Data.OracleClient 命名空间: 提供用于在 .NET 环境下访问 Oracle 数据库的 .NET Framework 数据提供程序。

在后面几节将详细介绍 System.Data.SqlClient 和 System.Data.OleDb 两个命名空间的使用, 关于 ODBC 和 Oracle 数据库的访问, 本书就不做进一步的讲解了。

24.2 DataSet 和 DataTable 类

DataSet 和 DataTable 类是 ADO.NET 核心, 它们用来保存和管理数据记录, 同时支持数据记录的添加、插入和修改等操作。本节以本地数据访问为实例, 详细介绍 DataSet 和 DataTable 类的使用。

24.2.1 表示内存数据表的 DataTable 类

DataTable 是 ADO.NET 库中的核心对象, 表示内存中的一个数据表。一个 DataTable 通过 DataColumnCollection 属性来表示表中的列结构, 如果 Table 从数据库获取, 则不需要设置列结构属性。但如果以编程方式创建 DataTable, 则必须先将多个 DataColumn 对象添加到 DataColumnCollection 中, 完成其列架构定义, 然后才能向表中添加数据记录。

通常, 使用 DataRow 向 DataTable 中添加一个数据记录, 必须先使用 DataTable 的 NewRow() 方法返回新的 DataRow 对象, 然后再设置新的 DataRow 的数据。一个 DataTable

可存储的最大行数是 16 777 216 的数据记录。

DataTable 也包含可用于确保数据完整性的 Constraint 对象的集合。另外,有许多 DataTable 相关事件可用于确定数据记录更改的时间,其中包括 RowChanged、RowChanging、RowDeleting 和 RowDeleted。表 24.2 列出了 DataTable 类的常用成员。

表 24.2 DataTable类的常用成员

分 类	名 称	访 问 性	说 明
属性	TableName	Public	读写属性, 获取或设置DataTable的名称
	IsInitialized	Public	只读属性, 指示是否已初始化DataTable
	HasErrors	Public	只读属性, 读取表所属DataSet的任何表的任何行中是否有错误
	CaseSensitive	Public	读写属性, 指示表中的字符串比较是否区分大小写
	ExtendedProperties	Public	只读属性, 获取自定义用户信息的集合
	MinimumCapacity	Public	读写属性, 获取或设置该表最初的起始大小, 表示可容纳的记录数
	Namespace	Public	读写属性, 获取或设置DataTable中所存储数据的XML, 表示形式的命名空间
	DataSet	Public	只读属性, 获取此表所属的DataSet
	Columns	Public	只读属性, 获取属于该表的列的集合
	PrimaryKey	Public	读写属性, 获取或设置充当数据表主键的列的数组
	Rows	Public	只读属性, 获取属于该表的行的集合
	DefaultView	Public	只读属性, 获取可能包括筛选视图或游标位置的表的自定义视图
	DisplayExpression	Public	读写属性, 获取或设置一个表达式, 该表达式返回的值用于表示用户界面中的此表
	ChildRelations	Public	只读属性, 获取此DataTable的子关系的集合
	ParentRelations	Public	只读属性, 获取该 DataTable 的父关系的集合
	Constraints	Public	只读属性, 获取由该表维护的约束的集合
方法	DataTable	Public	构造函数, 创建DataTable数据表对象
	BeginInit	Public	开始初始化在窗体上使用或由另一个组件使用的 DataTable
	EndInit	Public	结束在窗体上使用或由另一个组件使用的DataTable的初始化
	BeginLoadData	Public	在加载数据时关闭通知、索引维护和约束
	EndLoadData	Public	在加载数据后打开通知、索引维护和约束
	Load	Public	通过所提供的IDataReader, 用某个数据源的值填充 DataTable, 如果 DataTable已经包含行, 则从数据源传入的数据将与现有的行合并
	LoadDataRow	Public	查找和更新特定行, 如果找不到任何匹配行, 则使用给定值创建新行
	ImportRow	Public	将DataRow复制到DataTable中, 保留任何属性设置以及初始值和当前值
	Merge	Public	将指定的DataTable与当前的DataTable合并
	Copy	Public	复制该DataTable的结构和数据
	Clear	Public	清除DataTable中的所有记录数据

续表

分 类	名 称	访 问 性	说 明
方法	Reset	Public	将DataTable重置为其初始状态
	GetErrors	Public	获取包含错误的DataRow对象的数组
	GetChanges	Public	获取DataTable的更改副本, 它包含目前DataTable中所有未接受或拒绝的数据记录更改
	AcceptChanges	Public	提交目前DataTable中所有未接受或拒绝的数据记录更改
	RejectChanges	Public	回滚目前DataTable中所有未接受或拒绝的数据记录更改
	CreateDataReader	Public	返回与此DataTable中的数据相对应的DataTableReader
	NewRow	Public	创建与该表具有相同架构的新DataRow
	Select	Public	获取DataRow对象的数组
	ReadXml	Public	将XML架构和数据从XML数据流读入到DataTable
	WriteXml	Public	将DataTable的当前内容以XML格式写入到XML数据流
	ReadXmlSchema	Public	将XML架构从XML数据流读入到DataTable
	WriteXmlSchema	Public	将DataTable的当前数据结构以XML架构形式写入到XML数据流
事件	Initialized	Public	初始化DataTable后引发该事件
	TableNewRow	Public	插入新DataRow时引发该事件
	ColumnChanging	Public	在DataRow中指定的DataColumn的值发生更改时引发该事件
	ColumnChanged	Public	在DataRow中指定的DataColumn的值被更改后引发该事件
	RowChanging	Public	在DataRow正在更改时引发该事件
	RowChanged	Public	在成功更改DataRow之后引发该事件
	RowDeleting	Public	在表中的行要被删除之前引发该事件
	RowDeleted	Public	在表中的行已被删除后引发该事件
	TableClearing	Public	在清除DataTable时引发该事件
	TableCleared	Public	在清除DataTable后引发该事件

从表 24.2 中可以看出, DataTable 既可以通过 DataRow 添加数据, 也可以从 XML 数据流中读取数据, 它同样可以将数据写到 XML 文件中。

24.2.2 创建 DataTable 的方法

DataTable 表示一个数据表, 一般可以通过两种方式创建, 一是通过数据库操作从数据库获取, 二是直接通过 DataTable 的构造函数创建。第一种方法将在后面章节详细介绍, 本节主要介绍第二种方法。

通过 DataTable 的构造函数创建一个数据表, 通常需要下面 3 个步骤。

(1) 通过 DataTable 类的构造函数创建一个 DataTable 对象。DataTable 类的构造函数具有 3 个 public 重载版本, 定义如下所示。

❑ DataTable(): 创建一个表名为空字符串的数据表。

❑ DataTable (string name): 创建一个表名为 name 指定值的数据表。

❑ `DataTable (string name, string namespace)`: 创建一个表名为 `name` 指定值, XML 格式中命名空间为 `namespace` 指定值的数据表。

(2) 通过 `DataColumn` 类的构造函数创建数据列对象, 并依次添加到 `DataTable` 的 `Columns` 属性中, 从而得到数据表的结构。`DataColumn` 类常用的构造函数有以下 3 个版本。

❑ `DataColumn (string name, Type ty)`: 创建一个列名为 `name` 指定值, 类型为 `ty` 指定类型的数据列, `DataType` 可以是数据库支持的任意数据类型。

❑ `DataColumn (string name, Type ty, string expr)`: 创建一个列名为 `name` 指定值, 类型为 `ty` 指定类型的数据列。参数 `expr` 指定用于创建该列的表达式。

❑ `DataColumn (string name, Type ty, string expr, MappingType mty)`: 创建一个列名为 `name` 指定值, 类型为 `ty` 指定类型的数据列。参数 `expr` 指定用于创建该列的表达式, 参数 `mty` 表示该列映射到 XML 数据源的节点类型。

(3) 通过 `DataTable` 类的 `NewRow()` 方法获取符合当前表结构的 `DataRow` 对象, 并为该对象设置对应字段的数据, 然后将该 `DataRow` 对象添加到 `DataTable` 类的 `Rows` 属性中, 从而完成向 `DataTable` 中添加任意数量的数据记录。

下面代码演示了 `DataTable` 数据表的创建过程。其中, `CreateDataTable()` 方法创建数据表, 并按照固定格式添加指定数量的记录, `ShowDataTableProp()` 方法则通过 `DataTable` 类的常用属性获取并打印数据表的常用属性。

在 `CreateDataTable()` 方法中, 首先通过两种方式添加数据列 (`DataColumn`), 一种是通过 `DataColumn` 的构造函数创建对象 `col`, 然后添加到 `DataTable (dt)` 的 `Columns` 集合中。第二种是直接通过 `DataTable` 类的 `Columns` 属性的 `Add()` 方法添加。然后依次添加 `DataRow` 数据记录, 为 `DataRow` 对象设置数据有两种方式, 一种是通过列名来查找, 如 `row["年龄"] = (index * 10) % 100`。另外一种是通过从 0 开始的索引来查找, 如 `row[1] = "男"`。

```
using System.Data;

namespace _24._2._2
{
    class Program
    {
        static void Main(string[] args)
        {
            //创建数据表
            DataTable dt = CreateDataTable(6);
            //显示数据表的属性
            ShowDataTableProp(dt);
        }

        static DataTable CreateDataTable(int cnt)
        {
            //创建一个名为“用户”的数据表
            DataTable dt = new DataTable("用户");
            DataColumn col;
            //添加类型为字符串的“姓名”列
            col = new DataColumn("姓名",
                System.Type.GetType("System.String"));
            dt.Columns.Add(col);
            //添加类型为字符串的“性别”列
            dt.Columns.Add("性别", System.Type.GetType("System.String"));
        }
    }
}
```


```

//添加类型为整数的"年龄"列
col = new DataColumn("年龄", Type.GetType("System.Int32"));
dt.Columns.Add(col);

//添加数据记录到 DataTable
for (int index = 0; index < cnt; index++)
{
    //新建符合数据表格式的 DataRow 对象
    DataRow row = dt.NewRow();
    //设置"姓名"字段的值
    row["姓名"] = "姓名" + index.ToString("D2");
    //设置"性别"字段的值
    if (index % 2 == 0)
    {
        row[1] = "男";
    }
    else
    {
        row[1] = "女";
    }
    //设置"年龄"字段的值
    row["年龄"] = (index * 10) % 100;
    //添加到数据表
    dt.Rows.Add(row);
}
return dt;
}

//显示数据表的结构等属性
static void ShowDataTableProp(DataTable dt)
{
    //打印数据表的一些常用属性
    System.Console.WriteLine("数据表----{0}", dt.TableName);
    System.Console.WriteLine("\t 列数:{0}", dt.Columns.Count);
    System.Console.WriteLine("\t 行数:{0}", dt.Rows.Count);
    System.Console.WriteLine("\t 错误:{0}", dt.HasErrors);
    System.Console.WriteLine("\t 区分大小写:{0}", dt.CaseSensitive);
    System.Console.WriteLine("\t 是否初始化:{0}", dt.IsInitialized);
}
}
}

```

 **技巧:** 笔者建议通过列名来访问某一行数据记录中的指定字段的值,而不是通过从 0 开始的索引来访问,因为索引可能会变动,但是列名则不会发生变化。用列名访问可以放心调整数据表中列的顺序,不会造成错误。

生成并运行程序,得到的输出结果如下所示。

```

数据表----用户
列数:3
行数:6
错误:False
区分大小写:False
是否初始化:True

```


24.2.3 遍历 DataTable 中保存的记录

在实际开发过程中，当得到一个数据表（无论是从数据库获取，还是代码创建，或者从 XML 数据获取），对该数据表都有两个重要的操作，即查看数据表列结构、查看数据表记录。

在 `DataTable` 类中，可以通过 `Columns` 属性查看当前表中的列结构，`Columns` 表示 `DataColumn` 类的集合，可以用 `foreach` 遍历，也可以用“`[]`”操作符访问。可以通过列名和列索引两种形式访问指定列，通常在不知道列名的情况下使用列索引访问 `Columns` 属性中的列。

在 `DataTable` 类中，可以通过 `Rows` 属性查看当前表中所有的数据记录，`Rows` 表示 `DataRow` 类的集合，可以用 `foreach` 遍历，也可以用“`[]`”操作符访问。`DataRow` 类的 `RowState` 属性，表示数据记录的状态（例如添加、修改、删除等），`DataRow` 类的 `RowError` 属性表示当前记录是否有错误，比如，将不是整数的字符串（如 `abc`）赋值到“年龄”字段。

下面代码演示了 `DataTable` 类的 `Columns` 和 `Rows` 属性的使用，如何通过它们遍历数据表中所有的列和行数据。首先，通过 `Columns` 属性的 `Count` 属性获取列数，用从 0 开始的索引遍历每一列，并打印出列的信息，因为并不知道表中列的名称。然后通过 `Rows` 属性的 `Count` 属性获取记录数（行数），用从 0 开始的索引遍历每一条数据记录，并按列打印它的值。

```
using System.Data;

namespace _24._2._3
{
    class Program
    {
        static void Main(string[] args)
        {
            //创建数据表
            DataTable dt = CreateDataTable(6);
            //显示数据表记录
            ShowDataTable(dt);
        }
        //CreateDataTable 函数参见上例中的代码，为节约篇幅，这里不再列出

        //依次打印数据表的数据记录
        static void ShowDataTable(DataTable dt)
        {
            //打印数据表信息
            System.Console.WriteLine("数据表----{0}", dt.TableName);

            //依次打印数据表列信息
            System.Console.WriteLine("列信息--共{0}列", dt.Columns.Count);
            for (int colIndex = 0; colIndex < dt.Columns.Count; colIndex++)
            {
                //获取对应列
                DataColumn col = dt.Columns[colIndex];
                //打印数据列信息：索引，列名，标题，类型，是否允许空
                System.Console.WriteLine("\t列[{0}]:列名={1},标题={2},
```

```

        类型={3},允许空={4}",
        colIndex, col.ColumnName, col.Caption, col.DataType,
        col.AllowDBNull);
    }

    //依次打印数据表数据记录信息
    System.Console.WriteLine("记录信息--共{0}条记录", dt.Rows.Count);
    for (int rowIndex = 0; rowIndex < dt.Rows.Count; rowIndex++)
    {
        //获取对应数据记录
        DataRow row = dt.Rows[rowIndex];
        //打印数据记录的汇总信息
        System.Console.Write("\t行[{0}],状态:{1} ", rowIndex,
            row.RowState);
        //按列打印当前数据记录的所有数据
        for (int colIndex = 0; colIndex < dt.Columns.Count; colIndex++)
        {
            System.Console.Write(row[colIndex].ToString() + " ");
        }
        //打印换行
        System.Console.WriteLine();
    }
}
}
}

```

生成并运行代码，得到程序输出如下所示。由于在 `CreateDataTable()` 方法中所有行都是新增的，所以它们的状态都为新增（Added）。

```

数据表----用户
列信息--共 3 列
    列[0]:列名=姓名,标题=姓名,类型=System.String,允许空=True
    列[1]:列名=性别,标题=性别,类型=System.String,允许空=True
    列[2]:列名=年龄,标题=年龄,类型=System.Int32,允许空=True
记录信息--共 6 条记录
    行[0],状态:Added 姓名 00    男    0
    行[1],状态:Added 姓名 01    女   10
    行[2],状态:Added 姓名 02    男   20
    行[3],状态:Added 姓名 03    女   30
    行[4],状态:Added 姓名 04    男   40
    行[5],状态:Added 姓名 05    女   50

```

24.2.4 接受和回滚 DataTable 的更改

`DataTable` 类作为数据库表，必须支持两个基本的操作，即接受变化和回滚变化。接受变化是保存目前数据表中的数据记录变化，回滚变化则是取消目前数据表中的数据记录变化。接受变化和回滚变化不仅仅用于数据库，也可用在简单数据表格中。

在 `DataTable` 类中，通过成员方法 `AcceptChanges()` 接受当前表中所有变化，它不接受任何参数，也不返回任何值。执行过后，`DataTable` 类中所有 `DataRow` 的变化都被接受，且 `RowState` 属性都变成没有变化（`UnChanged`）。

在 `DataTable` 类中，通过成员函数 `RejectChanges()` 回滚当前表中所有变化，它不接受任何参数，同样也不返回任何值。执行过后，`DataTable` 类中所有 `DataRow` 的变化都被回

滚，即如果是新增行则被删除；如果是被修改行则恢复到修改前的值；如果是被删除的行则被重新添加回来，且所有 DataRow 的 RowState 属性都变成没有变化（UnChanged）。

有些时候，可能只希望接受或回滚某些特定条件的数据记录，这就需要使用 DataRow 类的 AcceptChanges() 和 RejectChanges() 方法，它们分别接受或回滚当前行的修改，如果没有修改则没有任何效果。

下面代码演示了如何通过 DataTable 类和 DataRow 类来接受或回滚数据记录的变化。方法 CreateDataTable() 创建包含指定数量数据记录的数据表，创建完成后所有的数据记录状态都为新增（Added）。首先，创建数据表 dt1，通过 DataTable 类的 AcceptChanges() 方法接受所有变化，并打印 dt1 中的所有行。然后创建数据表 dt2，通过 DataTable 类的 RejectChanges() 方法回滚所有变化，并打印 dt2 中的所有行。最后创建数据表 dt3，通过方法 AcceptChange() 接受 dt 中偶数行数据记录的变化，回滚剩下数据记录的变化，并打印 dt3 中的所有行。

```
using System.Data;

namespace _24._2._4
{
    class Program
    {
        static void Main(string[] args)
        {
            DataTable dt1, dt2, dt3;

            //接受全部变化
            dt1 = CreateDataTable(6);
            dt1.AcceptChanges();
            ShowDataRows(dt1);

            //回滚全部变化
            dt2 = CreateDataTable(8);
            dt2.RejectChanges();
            ShowDataRows(dt2);

            //接受部分变化，回滚部分变化
            dt3 = CreateDataTable(10);
            AcceptChange(dt3);
            ShowDataRows(dt3);
        }

        static DataTable CreateDataTable(int cnt)
        {
            //创建一个名为“用户”的数据表
            DataTable dt = new DataTable("用户" + cnt.ToString("D2"));
            DataColumn col;
            //添加类型为字符串的“姓名”列
            col = new DataColumn("姓名", System.Type.GetType("System.String"));
            dt.Columns.Add(col);
            //添加类型为字符串的“性别”列
            dt.Columns.Add("性别", System.Type.GetType("System.String"));
            //添加类型为整数的“年龄”列
```



```

col = new DataColumn("年龄", Type.GetType("System.Int32"));
dt.Columns.Add(col);

//添加数据记录到 DataTable
for (int index = 0; index < cnt; index++)
{
    //新建符合数据表格式的 DataRow 对象
    DataRow row = dt.NewRow();
    //设置"姓名"字段的值
    row["姓名"] = "姓名" + index.ToString("D2");
    //设置"性别"字段的值
    if (index % 2 == 0)
    {
        row[1] = "男";
    }
    else
    {
        row[1] = "女";
    }
    //设置"年龄"字段的值
    row["年龄"] = (index * 10) % 100;
    //添加到数据表
    dt.Rows.Add(row);
}
return dt;
}

static void ShowDataTableRows(DataTable dt)
{
    System.Console.WriteLine("数据表----{0}", dt.TableName);
    //依次打印数据表数据记录信息
    System.Console.WriteLine("记录信息--共{0}条记录", dt.Rows.Count);
    for (int rowIndex = 0; rowIndex < dt.Rows.Count; rowIndex++)
    {
        //获取对应数据记录
        DataRow row = dt.Rows[rowIndex];
        //打印数据记录的汇总信息
        System.Console.WriteLine("\t行[{0}],状态:{1} ", rowIndex,
            row.RowState);
        //按列打印当前数据记录的所有数据
        for (int colIndex = 0; colIndex < dt.Columns.Count; colIndex++)
        {
            System.Console.Write(row[colIndex].ToString() + " ");
        }
        //打印换行
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}

//接收部分变化
static void AcceptChange(DataTable dt)
{
    //接受数据表中偶数行的变化
    for (int i = 0; i < dt.Rows.Count; i++)

```



```

        {
            if (i % 2 == 0)
            {
                dt.Rows[i].AcceptChanges();
            }
        }
        //其他变化全部回滚
        dt.RejectChanges();
    }
}

```

运行代码，得到的输出如下所示。其中 dt1 所有的数据记录（共 6 条）都被接受并保存在数据表 dt1 中。dt2 中所有的数据记录（共 8 行）都被回滚，所以 dt2 中没有任何数据记录。dt3 中的数据记录被部分接受，所以只有偶数部分数据记录还留在 dt3 中，其他都被回滚。


数据表----用户 06
记录信息--共 6 条记录

行[0],状态:Unchanged	姓名 00	男	0
行[1],状态:Unchanged	姓名 01	女	10
行[2],状态:Unchanged	姓名 02	男	20
行[3],状态:Unchanged	姓名 03	女	30
行[4],状态:Unchanged	姓名 04	男	40
行[5],状态:Unchanged	姓名 05	女	50

数据表----用户 08
记录信息--共 0 条记录

数据表----用户 10
记录信息--共 5 条记录

行[0],状态:Unchanged	姓名 00	男	0
行[1],状态:Unchanged	姓名 02	男	20
行[2],状态:Unchanged	姓名 04	男	40
行[3],状态:Unchanged	姓名 06	男	60
行[4],状态:Unchanged	姓名 08	男	80

 **技巧：** DataTable 类除了用在数据库记录存储之外，还可以用于普通数据的保存。利用它方便的接受和回滚能力，可以轻松维护一个完整、清洁的数据表。

24.2.5 表示内存数据集合的 DataSet 类

和 DataTable 一样，DataSet 也是 ADO.NET 的核心对象，DataSet 类用来表示内存中的数据集合，可以将它看成是一个简单的内存数据库。一个 DataSet 可以包含多个 DataTable，并且可以包含 DataTable 之间的关系、限制等信息。另外，DataSet 同样可以表示 XML 数据，可以从 XML 数据流读取数据到 DataSet，也可以将 DataSet 中的数据写入到 XML 数据流中。

DataSet 中所有的数据表都可以通过它的 Tables 属性访问，DataSet 中数据的存储实际是通过 DataTable 来实现。表 24.3 给出了 DataSet 类的常用成员。


表 24.3 DataSet类常用成员

分 类	名 称	访 问 性	说 明
属性	DataSetName	Public	读写属性, 表示当前DataSet的名称
	Tables	Public	只读属性, 表示包含在DataSet中的表(DataTable)的集合
	CaseSensitive	Public	读写属性, 表示DataTable对象中的字符串比较是否区分大小写
	HasErrors	Public	只读属性, 表示在DataSet中的任何DataTable对象中是否存在错误
	IsInitialized	Public	只读属性, 表示是否初始化DataSet
	DefaultViewManager	Public	只读属性, 获取DataSet所包含的数据的自定义视图, 以允许使用自定义的DataViewManager进行筛选、搜索和导航
	Relations	Public	只读属性, 获取将表链接起来并允许从父表浏览到子表的关系的集合(数据库关系集合)
	EnforceConstraints	Public	读写属性, 表示在尝试执行任何更新操作时是否遵循约束规则
	ExtendedProperties	Public	只读属性, 获取与DataSet相关的自定义用户信息的集合
	Namespace	Public	读写属性, 表示DataSet的命名空间
	Prefix	Public	读写属性, 表示作为XML数据时, DataSet的命名空间的别名
	Site	Public	读写属性, 获取或设置DataSet的System.ComponentModel.ISite
方法	DataSet	Public	构造函数, 创建具有指定属性的DataSet对象
	HasChanges	Public	获取一个值, 该值指示DataSet是否有更改, 包括新增行、已删除的行或已修改的行
	GetChanges	Public	获取DataSet中被修改过的数据记录的集合副本
	AcceptChanges	Public	提交当前数据集中所有未接受或拒绝的更改
	RejectChanges	Public	回滚当前数据集中所有未接受或拒绝的更改
	BeginInit	Public	开始初始化在窗体上使用或由另一个组件使用的DataSet
	EndInit	Public	结束在窗体上使用或由另一个组件使用的DataSet的初始化
	Clear	Public	清除DataSet中的所有数据, 通过清除所有表中的所有行实现
	Reset	Public	将DataSet重置为其初始状态
	Copy	Public	复制当前DataSet的结构和数据
	Merge	Public	将新的数据合并到当前DataSet或DataTable中, 这些数据可以是一个DataSet、DataTable或DataRow的数组
	CreateDataReader	Public	为每个DataTable创建一个带有结果集的DataTableReader对象, 可以只读向前访问数据表中的记录
	Load	Public	通过所提供的IDataReader, 用某个数据源的值填充DataSet
	GetXml	Public	返回存储在DataSet中的数据的XML表示形式

续表

分 类	名 称	访 问 性	说 明
方法	GetXmlSchema	Public	返回存储在DataSet中的数据XML表示形式的XML架构
	InferXmlSchema	Public	将XML架构应用于DataSet
	ReadXml	Public	将XML架构和数据读入DataSet
	ReadXmlSchema	Public	将XML架构读入DataSet
	WriteXml	Public	往DataSet写XML数据，还可以选择写架构
	WriteXmlSchema	Public	写XML架构形式的DataSet结构

从表 24.3 可以看出，DataSet 类的一些成员函数从名称到功能上都与 DataTable 相似，如 AcceptChanges、RejectChanges 等，这些方法通常是通过 DataSet 中的各 DataTable 执行对应函数完成，比如 DataSet 类的 AcceptChanges()方法就是依次执行 DataSet 中所有 DataTable 的 AcceptChanges()方法来实现的。

说明：DataTable 类的重点在于数据记录（DataRow）的管理，DataSet 类的重点则在于数据表（DataTable）的管理，比如，Copy()、Merge()等方法实现对数据表中数据记录的复制、合并等。另外，DataSet 类还在 XML 数据支持上实现了完整的功能（本章暂不讨论这方面的内容）。

24.2.6 使用 DataSet 类的步骤

DataSet 类在实际开发中，常用来在内存中保存多数据表、相互之间还可能具有相互约束等关系的数据记录。在使用 DataSet 类之前，首先要创建一个 DataSet 类的对象，DataSet 类的使用大致包括以下 3 个步骤。

（1）创建 DataSet 类对象，可以通过 DataSet 类的构造函数创建 DataSet 对象实例，DataSet 构造函数具有两个常用版本，定义如下所示。

- ❑ DataSet(): 创建一个不带任何参数的 DataSet 对象，数据集的名称为默认值 NewDataSet。
- ❑ DataSet(string name): 创建一个具有指定名称的 DataSet 对象，参数 name 指定新数据集的名称。

（2）DataSet 对象创建好之后，它仅仅是一个空的数据集，不包含任何数据。因此，需要为 DataSet 对象添加一个或多个数据表，这些都通过 DataSet 类的 Tables 属性完成。如果需要，还可以为 DataSet 中所有的 DataTable 添加关系。

下面的代码演示了如何创建 DataSet 类对象，其中，CreateDataSet()方法首先通过 DataSet 类构造函数创建名为“用户数据集”的 DataSet 对象 ds。然后通过 DataSet 类的 Tables 属性将两个数据表（用户编号和用户信息）添加到 ds 中，两个表之间用“编号”字段联系起来。ShowDataSetProp()方法则是通过 DataSet 类的属性访问它的属性，并打印。另外，该方法还通过 DataSet 类的 Tables 属性遍历 DataSet 中的所有数据表，并打印出 DataTable 的列结构。

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using System.Data;

namespace _24._2._6
{
    class Program
    {
        static void Main(string[] args)
        {
            //创建数据集合
            DataSet ds = CreateDataSet( );
            //显示数据集的信息
            ShowDataSetProp(ds);
        }

        static DataSet CreateDataSet( )
        {
            DataSet ds;
            DataTable dt;
            DataColumn col;

            //创建数据集合对象 ds
            ds = new DataSet("用户数据集");
            //创建第1个数据表—用户编号
            dt = new DataTable("用户编号");
            //创建数据列--编号, 并添加到数据表
            col = new DataColumn("编号", Type.GetType("System.String"));
            col.AllowDBNull = false;
            dt.Columns.Add(col);
            //创建数据列--密码, 并添加到数据表
            col = new DataColumn("密码", Type.GetType("System.String"));
            col.AllowDBNull = false;
            dt.Columns.Add(col);
            //将数据表添加到数据集合 ds
            ds.Tables.Add(dt);

            //创建第2个数据表—用户信息
            dt = new DataTable("用户信息");
            //创建数据列--编号, 并添加到数据表
            col = new DataColumn("编号", Type.GetType("System.String"));
            col.AllowDBNull = false;
            dt.Columns.Add(col);
            //创建数据列--姓名, 并添加到数据表
            col = new DataColumn("姓名", Type.GetType("System.String"));
            col.AllowDBNull = false;
            dt.Columns.Add(col);
            //创建数据列--年龄, 并添加到数据表
            col = new DataColumn("年龄", Type.GetType("System.Int32"));
            col.AllowDBNull = false;
            dt.Columns.Add(col);
            //创建数据列--电话, 并添加到数据表
            col = new DataColumn("电话", Type.GetType("System.String"));
            col.AllowDBNull = true;
            dt.Columns.Add(col);
            //将数据表添加到数据集合 ds
            ds.Tables.Add(dt);

            return ds;
        }
    }
}
```



```

    }

    static void ShowDataSetProp(DataSet ds)
    {
        //显示 DataSet 的汇总信息
        System.Console.WriteLine("DataSet--{0}", ds.DataSetName);
        System.Console.WriteLine(
            "  包含{0}个表, 区分大小写={1}, 有错误={2}, 有变化={3}",
            ds.Tables.Count, ds.CaseSensitive, ds.HasErrors,
            ds.HasChanges());
        //依次显示 DataSet 中 DataTable 的具体信息
        foreach (DataTable dt in ds.Tables)
        {
            System.Console.WriteLine("  DataTable--{0}, {1}列, {2}行",
                dt.TableName, dt.Columns.Count,
                dt.Rows.Count);
            //依次显示 DataTable 中所有 DataColumn 的具体信息
            for (int colIndex = 0; colIndex < dt.Columns.Count; colIndex++)
            {
                DataColumn col = dt.Columns[colIndex];
                System.Console.WriteLine(
                    "    Columns[{0}]:Name={1}, Type={2}, AllowNULL={3}",
                    colIndex, col.ColumnName,
                    col.DataType.ToString(),
                    col.AllowDBNull);
            }
        }
    }
}

```

运行代码得到如下输出, 由于没有添加数据, 所以 DataSet 对象 ds 中还没有错误, 也没有任何变化发生。

DataSet--用户数据集

包含 2 个表, 区分大小写=False, 有错误=False, 有变化=False

DataTable--用户编号, 2 列, 0 行

Columns[0]:Name=编号, Type=System.String, AllowNULL=False

Columns[1]:Name=密码, Type=System.String, AllowNULL=False

DataTable--用户信息, 4 列, 0 行

Columns[0]:Name=编号, Type=System.String, AllowNULL=False

Columns[1]:Name=姓名, Type=System.String, AllowNULL=False

Columns[2]:Name=年龄, Type=System.Int32, AllowNULL=False

Columns[3]:Name=电话, Type=System.String, AllowNULL=True

(3) 为 DataSet 类添加 DataTable 之后, 需要为其中的 DataTable 添加数据记录, 可以直接通过 Tables 中的某个 DataTable 对象添加。当然, DataTable 中的数据也可以在创建 DataTable 时添加。

在下面的代码中, AddUserInfo() 方法通过向 DataTable 类添加数据记录, 实现向 DataSet 的数据表中添加单条数据记录。从代码中可以看出, 可以通过两种方法访问 DataSet 中的 DataTable, 第一种是通过表名获取, 如 dt = ds.Tables["用户编号"], 第二种是通过表索引获取, 如 dt = ds.Tables[1]。

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Data;

namespace _24._2._6
{
    class Program
    {
        static void Main(string[] args)
        {
            //创建数据集合
            DataSet ds = CreateDataSet( );
            //添加 2 条数据记录
            AddUserInfo(ds, "001", "pwd", "张三", "13612345678", 20);
            AddUserInfo(ds, "005", "mima", "李四", "13612345679", 25);
            //显示数据集的数据记录
            ShowDataSetRows(ds);
        }
        // CreateDataSet 函数的代码参见上一例

        static void AddUserInfo(DataSet ds, string id, string pwd,
            string name, string tel, int age)
        {
            DataTable dt;
            DataRow dr;
            //通过表名获取“用户编号”表
            dt = ds.Tables["用户编号"];
            //向用户编号表添加一行
            dr = dt.NewRow( );
            dr["编号"] = id;
            dr["密码"] = pwd;
            dt.Rows.Add(dr);

            //用表索引获取“用户信息”表
            dt = ds.Tables[1];
            //向用户信息表添加一行
            dr = dt.NewRow( );
            dr["编号"] = id;
            dr["姓名"] = name;
            dr["年龄"] = age;
            dr["电话"] = tel;
            dt.Rows.Add(dr);
        }

        static void ShowDataSetRows(DataSet ds)
        {
            //显示 DataSet 的汇总信息
            System.Console.WriteLine("DataSet--{0}", ds.DataSetName);
            System.Console.WriteLine(
                "  包含{0}个表, 区分大小写={1}, 有错误={2}, 有变化={3}",
                ds.Tables.Count, ds.CaseSensitive,
                ds.HasErrors, ds.HasChanges( ));
            //依次显示 DataSet 中 DataTable 的具体信息
            foreach (DataTable dt in ds.Tables)
            {
                System.Console.WriteLine("  DataTable--{0}, {1}列, {2}行",
                    dt.TableName, dt.Columns.Count, dt.

```



```
        Rows.Count);  
    //依次显示 DataTable 中所有每条数据记录  
    foreach (DataRow dr in dt.Rows)  
    {  
        //依次打印各列记录  
        for (int colIndex = 0; colIndex < dt.Columns.Count;  
            colIndex++)  
        {  
            System.Console.Write("      " + dr[colIndex].  
                ToString());  
        }  
        //打印换行  
        System.Console.WriteLine( );  
    }  
}  
}
```

运行代码，可以得到程序输出如下所示。由于添加了新数据，所以 DataSet 中有变化发生，因此 HasChanges()方法返回 true。

```
DataSet--用户数据集
    包含 2 个表，区分大小写=False，有错误=False，有变化=True
    DataTable--用户编号，2 列，2 行
        001      pwd
        005      mima
    DataTable--用户信息，4 列，2 行
        001      张三      20      13612345678
        005      李四      25      13612345679
```

24.2.7 接受和回滚 DataSet 的更改

和 DataTable 一样，通过 AcceptChanges()方法来接受 DataSet 中所有数据表中所有的更改，通过 RejectChanges()方法回滚 DataSet 中所有数据表中所有的更改。执行完成后 DataSet 类的 HasChanges()方法将返回 false，因为不存在未提交或回滚的更改。

有时，不需要接受或回滚 DataSet 中所有数据记录中的更改，则可以通过 DataSet 的 Tables 属性获取要接受或回滚的 DataTable，通过 24.2.4 节中的方法独立接受或回滚数据表的更改记录。执行完成后，DataSet 类的 HasChanges() 方法的返回值将根据实际情况返回 true 或 false，因为其他表可能还包含更改记录。

下面代码中，AcceptAllChanges()方法通过 DataSet 类的 AcceptChanges()方法接受 DataSet 中的所有更改记录。RejectAllChanges()方法通过 DataSet 类的 RejectChanges()方法回滚 DataSet 中的所有更改记录。AcceptPartChanges()方法首先通过 DataTable 类的 AcceptChanges()方法接受“用户编号”表的更改记录，然后通过 DataTable 类的 RejectChanges()方法回滚“用户信息”表的更改记录。

```
static void AcceptAllChanges(DataSet ds)
{
    //通过 DataSet 类的 AcceptChanges() 方法接受所有更改
    ds.AcceptChanges();
}
```

```
static void RejectAllChanges(DataSet ds)
{
    //通过 DataSet 类的 AcceptChanges() 方法接受所有更改
    ds.RejectChanges();
}

static void AcceptPartChanges(DataSet ds)
{
    DataTable dt;
    //通过 DataTable 类的 AcceptChanges() 方法接受表 用户编号 的更改
    dt = ds.Tables["用户编号"];
    dt.AcceptChanges();
    //通过 DataTable 类的 RejectChanges() 方法回滚表 用户信息 的更改
    dt = ds.Tables["用户信息"];
    dt.RejectChanges();
}
```

24.3 用 ADO.NET 访问 SQL Server 数据库

SQL Server 数据库是中小应用软件中的主流数据库，配置简单，使用方便。本节将介绍如何通过 ADO.NET 访问 SQL Server 数据库，并插入、添加、更改数据库记录。

24.3.1 ADO.NET 访问数据库的步骤

ADO.NET 实现对多种数据库访问支持，虽然后台数据库可能是不同类型，例如，SQL Server、Access、Oracle 等，但是通过 ADO.NET 访问数据库的模式和基本步骤都是类似的。

ADO.NET 提供两种模式访问数据库：有连接和无连接。在有连接模式下访问数据库，在取得数据库连接后，保持数据库连接，通过向数据库服务器发送 SQL 命令等方式实时更新数据库。在有连接模式下的数据库访问通常包括以下步骤：

(1) 通过数据库连接类（Connection）连接到数据库，如 SQL Server 服务器、Access 数据库文件等。

(2) 通过数据库命令类（Command）在数据库上执行 SQL 语句，可以是任何 SQL 语句，包括更新（Update）、插入（Insert Inot）、删除（Delete）、查询（Select）等。

(3) 如果是查询语句，还可以通过数据读取器（DataReader）类进行只读只向前读取数据记录。

(4) 数据库操作完成后通过连接类（Connection）关闭数据库连接。

有连接模式下进行数据库访问，尽量不要长时间操作，因为这样会导致数据库服务器被长期占用，影响其他客户端连接到数据库服务器。笔者建议在使用之前打开数据库连接，使用完成后马上关闭数据库连接。

在需要对数据进行长时间处理时，通常采用无连接模式进行数据访问。无连接模式下，需要处理的数据库服务器中的数据在本地有一个副本，通常保存在 DataSet 或 DataTable 中，ADO.NET 通过数据适配器（DataAdapter）将本地数据和数据库服务器关联起来。在从数据库服务器获取到数据之后，数据适配器断开与服务器的连接，对数据的修改通过修改本

地 DataSet 完成,然后再通过数据适配器更新到服务器。在 ADO.NET 中,无连接模式的数据库访问通常需要以下步骤:

- (1) 通过数据库连接类 (Connection) 指定需要连接的数据库服务器。
- (2) 创建基于该数据库连接的数据适配器,并指定更新数据库的语句,包括插入 (Insert)、更新 (Update)、查询 (Select)、删除 (Delete) 4 个命令。DataAdapter 通过这几个命令从数据库获取数据,也将本地的数据更改更新到数据库服务器中。
- (3) 通过 DataAdapter 从数据库服务器获取数据到本地 DataSet 或 DataTable 中。
- (4) 使用或更改本地 DataSet 或 DataTable 中的数据。
- (5) 通过 DataAdapter 将本地数据更改更新到数据库服务器,并关闭数据库连接。

基于无连接的数据库访问,具有执行效率高、数据库连接占用时间短、修改记录易更改和回滚等优点,但是也在一定程度上导致数据更新不及时。

24.3.2 用 SqlConnection 连接数据库

ADO.NET 支持多种数据库的访问,由于不同数据库具有不同的差别,只是在访问模式上进行了统一,访问不同的类型数据库还是需要不同的类来实现。在 ADO.NET 中主要分成 4 类数据库支持,即 ODBC、OleDb、SQL Server 和 Oracle。其中 ODBC 数据源已经逐渐被淘汰,而 Oracle 多用于大型数据处理应用软件开发,所以本章不介绍这两种数据库的访问。

ADO.NET 中,访问 SQL Server 数据库相关的类都封装在 System.Data.SqlClient 命名空间下,它主要包含以下几个类。

- ❑ SqlConnection: SQL Server 数据库连接类,通过它可以连接到指定的 SQL Server 数据库,并且提供打开和关闭数据库连接功能。
- ❑ SqlCommand: SQL Server 命令类,它表示一个可执行的 SQL 命令,可以是普通的文本 SQL 命令,也可以是带参数的 SQL 命令。SQL 命令可以通过 SqlCommandBuilder 类创建。
- ❑ SqlDataAdapter: SQL Server 数据适配器类,用于连接 SQL Server 数据库与本地 DataSet 和 DataTable 数据集,提供获取服务器数据、更新本地更改等操作。
- ❑ SqlDataReader: SQL Server 数据读取器,提供只读只向前的数据记录读取。

本节重点介绍 SqlConnection 类的使用,它表示一个 SQL Server 数据库的一个唯一会话,它通过指定的数据库连接字符串连接到数据库,并打开数据库。表 24.4 给出了 SqlConnection 类的常用成员。

表 24.4 SqlConnection 类常用成员

分 类	名 称	访 问 性	说 明
属性	ConnectionString	Public	读写属性,表示用于打开 SQL Server 数据库的连接字符串,包括数据库服务器的 IP 地址、端口、目标数据库、安全性等信息
	ConnectionTimeout	Public	只读属性,表示尝试连接到数据库服务器判断为连接失败的等待时间,单位为秒,由连接字符串指定

续表

分 类	名 称	访 问 性	说 明
属性	Database	Public	只读属性,表示当前数据库或连接打开后要使用的数据库的名称,由连接字符串指定
	DataSource	Public	只读属性,表示要连接的SQL Server实例的名称
	PacketSize	Public	只读属性,表示用来与SQL Server实例通信的网络数据包的大小,单位为字节
	ServerVersion	Public	只读属性,获取包含客户端连接的SQL Server实例的版本
	State	Public	只读属性,表示当前数据库连接的状态
	WorkstationId	Public	只读属性,获取标识数据库客户端的一个字符串
	StatisticsEnable	Public	读写属性,表示是否进行统计信息收集
方法	Open	Public	使用ConnectionString所指定的属性打开数据库连接
	Close	Public	关闭与数据库的连接
	BeginTransaction	Public	开始在SQL Server数据库上执行一个事务
	ChangeDatabase	Public	为打开的数据库连接更改当前数据库
	ChangePassword	Public	将连接字符串中指定用户的SQL Server密码更改为提供的新密码
	CreateCommand	Public	创建并返回一个与SqlConnection关联的SqlCommand对象
	ClearAllPools	Public	清空连接池
	ClearPool	Public	清空与指定连接关联的连接池

在表 24.4 列出的众多属性和方法中,在数据库连接和断开操作中最常用的有以下 3 个。

- ❑ **ConnectionString**: 连接字符串,它包含数据库服务器的地址、端口、目标数据库、连接超时时间、安全性、登录用户名和密码等信息。在进行数据连接之前,必须指定正确的连接字符串。
- ❑ **Open()**: 用于打开由 **ConnectionString** 属性指定的数据库连接,如果连接字符串不正确,或目标服务器不可用(比如没有打开、不存在等)都会抛出异常。
- ❑ **Close()**: 关闭一个已经打开的数据库连接,如果当前并没有连接,则不做任何操作。

下面代码中演示了如何通过 **SqlConnection** 类连接到数据库。其中, **ShowConnectionProp()** 方法首先通过 **SqlConnection** 类的构造函数创建具有指定 **ConnectionString** 的数据库连接。然后通过 **SqlConnection** 类的属性访问并打印出数据库连接的属性。接下来通过 **Open()** 方法打开数据库连接,并打印出数据库连接状态和 SQL Server 服务器版本。最后关闭数据库连接,并打印出数据库连接状态。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace _24._3._2
{
    class Program
    {
        static void Main(string[] args)
        {
```



```

        ShowConnectionProp();
    }

    static void ShowConnectionProp()
    {
        //数据库连接字符串
        string conStr = "Data Source=.\SQLEXPRESS;Initial Catalog=mydb;Integrated Security=True";
        //用连接字符串创建数据库连接对象
        SqlConnection con = new SqlConnection(conStr);
        //在连接之前打印连接对象各属性信息
        System.Console.WriteLine("数据库连接--Open() 前信息:");
        System.Console.WriteLine("  ConnectionString=[{0}]",
            con.ConnectionString);
        System.Console.WriteLine("  ConnectionTimeout=[{0}]",
            con.ConnectionTimeout);
        System.Console.WriteLine("  Database=[{0}]", con.Database);
        System.Console.WriteLine("  DataSource=[{0}]",
            con.DataSource);
        System.Console.WriteLine("  PacketSize=[{0}]",
            con.PacketSize);
        System.Console.WriteLine("  StatisticsEnabled=[{0}]",
            con.StatisticsEnabled);
        System.Console.WriteLine("  WorkstationId=[{0}]",
            con.WorkstationId);
        System.Console.WriteLine("  State=[{0}]", con.State);
        System.Console.WriteLine();
        //通过 Open() 连接到数据库
        con.Open();
        //打印连接后数据库连接信息, 并获取 SQL Server 服务器版本号
        System.Console.WriteLine("数据库连接--Open() 后信息:");
        System.Console.WriteLine("  ServerVersion=[{0}]",
            con.ServerVersion);
        System.Console.WriteLine("  State=[{0}]", con.State);
        System.Console.WriteLine();
        //通过 Close() 关闭数据库连接
        con.Close();
        //打印关闭后数据库连接信息
        System.Console.WriteLine("数据库连接--Close() 后信息:");
        System.Console.WriteLine("  State=[{0}]", con.State);
    }
}

```

以上代码中连接的数据库是本书第 23 章中创建的 SQL Server 数据库 mydb, 若无此数据库可参见第 23 章的步骤进行创建。连接字符串中的“Data Source=.\SQLEXPRESS”表示连接到本机的 SQL Server 2008 Express 服务器, 若不是 Express 版, 则不需要使用“SQLEXPRESS”, 直接写为“.”即可。另外, 这里的“.”表示当前服务器, 若连接到其他服务器的 SQL Server 服务器, 则需要将这个“.”替换为对应的服务器名称。

运行代码, 得到程序输出如下所示。Open()前和 Close()之后数据库连接状态都为关闭(Closed), 执行 Open()方法之后, 数据库连接状态变为打开(Open)。

```

数据库连接--Open() 前信息:
ConnectionString=[Data Source=.\SQLEXPRESS;Initial Catalog=mydb;
Integrated Security=True]
ConnectionTimeout=[15]

```

```

Database=[mydb]
DataSource=[.\SQLEXPRESS]
PacketSize=[8000]
StatisticsEnabled=[False]
WorkstationId=[WYHNPC]
State=[Closed]

```

数据库连接--Open()后信息:

```

ServerVersion=[10.00.2531]
State=[Open]


```

数据库连接--Close()后信息:

```

State=[Closed]

```

 注意: SqlConnection 类的 ServerVersion 属性只能在数据库连接打开时才能正确执行, 如果数据库连接没有打开, 则会抛出异常。

24.3.3 用 SqlCommand 执行 SQL 命令

在有连接数据库访问模式下, 通过 SqlConnection 类连接到 SQL Server 数据库服务器之后, 需要对服务器中的数据记录进行操作, 这就需要执行 SQL 命令。在 ADO.NET 中, 通过 SqlCommand 类实现 SQL Server 所支持的 SQL 命令。

SqlCommand 类封装了要对 SQL Server 数据库执行的 Transact-SQL 语句或存储过程。最常用也是最简单的使用方式是通过 SqlCommand 类直接执行表示 SQL 命令字符串。SqlCommand 类支持任何 SQL 命令语句的执行, 表 24.5 列出了它的常用成员。

表 24.5 SqlCommand 类常用成员

分 类	名 称	访 问 性	说 明
属性	CommandText	Public	读写属性, 表示要对数据源执行的 Transact-SQL 语句或存储过程
	CommandTimeout	Public	读写属性, 表示判断执行失败所需要的超时时间, 单位秒
	CommandType	Public	读写属性, 表示如何解释 CommandText 属性, 通常为文本格式
	Connection	Public	读写属性, 表示 SqlCommand 中要使用的数据库连接
	Transaction	Public	读写属性, 表示将在其中执行 SqlCommand 的事务
方法	BeginExecuteNonQuery	Public	启动一个异步操作, 执行 Transact-SQL 语句或存储过程指定操作, 通常为不返回数据集的操作, 比如 Update、Insert、Delete, 返回被影响的数据记录的条数
	ExecuteNonQuery	Public	同步执行 Transact-SQL 语句或存储过程指定操作, 通常为不返回数据集的操作, 比如 Update、Insert、Delete, 返回被影响的数据记录的条数
	EndExecuteNonQuery	Public	停止 BeginExecuteNonQuery() 启动的异步操作
	BeginExecuteReader	Public	启动一个异步操作, 执行 Transact-SQL 语句或存储过程指定的查询操作, 比如 Select 语句, 返回只读只向前的数据读取器

续表

分 类	名 称	访 问 性	说 明
方法	ExecuteReader	Public	同步执行Transact-SQL语句或存储过程指定的查询操作, 比如Selecte语句, 返回只读只向前的数据读取器
	EndExecuteReader	Public	停止BeginExecuteReader()启动的异步操作
	ExecuteScalar	Public	同步执行一个查询操作, 并返回查询所返回的结果集中第1行的第1列的数据, 忽略其他列或行
	Cancel	Public	尝试取消SqlCommand的执行操作
	ResetCommandTimeout	Public	将CommandTimeout属性重置为其默认值

通过 SqlCommand 类更改数据库记录, 通常分为两类操作: 更新类和查询类。更新类不需要返回数据集, 如 Update、Delete、Insert 命令, 通常通过 SqlCommand 类的 ExecuteNonQuery()方法执行这类命令, 返回实际影响的数据记录的数量。查询类则需要返回数据, 比如, Select 命令, 通常通过 SqlCommand 类的 ExecuteReader()方法执行命令, 并返回一个只读只向前的 DataReader。

通过 SqlCommand 类执行 SQL 命令通常包括以下几个步骤。

(1) 通过 SqlConnection 类建立可用的数据库连接。

(2) 在可用数据库连接基础上创建一个 SqlCommand 对象, 并通过 CommandText 属性设置它要执行的 SQL 命令。

(3) 打开数据库连接, 使用 Execute 系列方法执行 SQL 命令, 并对执行结果进行处理, 比如依次读取查询结果等。

(4) 关闭数据库连接。

下面代码演示了 SqlCommand 类的具体使用, 在 UseSqlCommand()方法中, 向 Students 数据库的 Major 表中添加 3 条数据, 并通过 Update 进行修改, 最后通过 Select 语句查询表中的数据记录, 并依次打印。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace _24._3._3
{
    class Program
    {
        static void Main(string[] args)
        {
            //ShowConnectionProp();
            UseSqlCommand( );
        }

        static void UseSqlCommand( )
        {
            //数据库连接字符串
            string conStr = "Data Source=.\SQLEXPRESS;Initial
Catalog=mydb;Integrated Security=True";
            //用连接字符串创建数据库连接对象
            SqlConnection con = new SqlConnection(conStr);
```

```

//创建 SQL 命令对象, 并设置它的数据库连接
SqlCommand cmd = new SqlCommand( );
cmd.Connection = con;

//通过 Open() 连接到数据库
con.Open( );

//添加 3 条记录到 Major 表中
cmd.CommandText = "INSERT INTO Major VALUES('计算机科学与技术',
'微积分(一)')";
cmd.ExecuteNonQuery( );
cmd.CommandText = "INSERT INTO Major VALUES('计算机科学与技术',
'微积分(二)')";
cmd.ExecuteNonQuery( );
cmd.CommandText = "INSERT INTO Major VALUES('计算机软件', 'Java
程序设计')";
cmd.ExecuteNonQuery( );

//查询数据表中所有的数据记录, 并打印
cmd.CommandText = "SELECT * FROM Major";
SqlDataReader dr = cmd.ExecuteReader( );
System.Console.WriteLine("\nMajor 表的数据记录(1)");
//依次读出所有的数据记录, 并显示到界面
while (dr.Read( ))
{
    System.Console.WriteLine( );
    for (int index = 0; index < dr.FieldCount; index++)
    {
        System.Console.Write(" {0}", dr[index].ToString( ));
    }
}
dr.Close( );

//“计算机软件”专业重命名为“计算机应用软件”
cmd.CommandText = "UPDATE Major SET major =
                    '计算机应用软件' WHERE major='计算机软件'";
cmd.ExecuteNonQuery( );

//查询数据表中所有的数据记录, 并打印
cmd.CommandText = "SELECT * FROM Major";
dr = cmd.ExecuteReader( );
System.Console.WriteLine("\nMajor 表的数据记录(2)");
//依次读出所有的数据记录, 并显示到界面
while (dr.Read())
{
    System.Console.WriteLine( );
    for (int index = 0; index < dr.FieldCount; index++)
    {
        System.Console.Write(" {0}", dr[index].ToString());
    }
}
dr.Close( );

//通过 Close() 关闭数据库连接
con.Close( );
}
}
}

```


为了运行以上程序，需在数据库 mydb 中增加一个表格 Major，该表有两列，分别是 Major（专业）和 Course（课程）。

运行代码，得到输出如下，其中，前两条数据记录都是通过 Insert 语句添加，第 3 条记录首先通过 Insert 语句添加，然后又通过 Update 语句对它的“专业”字段进行修改。

```
Major 表的数据记录 (1)
  计算机科学与技术 微积分 (二)
  计算机科学与技术 微积分 (一)
  计算机软件 Java 程序设计
Major 表的数据记录 (2)
  计算机科学与技术 微积分 (二)
  计算机科学与技术 微积分 (一)
  计算机应用软件 Java 程序设计
```

24.3.4 用 SqlDataReader 读取数据库记录

在基于连接的数据库访问模式下，数据记录的读取通常通过 DataReader 完成。SqlDataReader 类是 ADO.NET 提供的用于读取 SQL Server 数据库记录的只读只向前数据记录读取器。

在 24.3.3 节的代码中，已经给出 SqlDataReader 类的简单使用实例，其中 Read() 方法是将数据记录读取器的当前记录指针移到下一条记录，直到全部读取完成。在当前数据记录的基础上，SqlDataReader 类还提供了很多 GetXXXX() 系列方法，将指定字段的数据按照某种数据类型读取，比如 int、string 等。表 24.6 给出了 SqlDataReader 类的常用成员。

表 24.6 SqlDataReader 类常用成员

分 类	名 称	访 问 性	说 明
属性	Depth	Public	只读属性，表示当前行的嵌套深度
	FieldCount	Public	只读属性，表示当前行中的列数
	HasRows	Public	只读属性，表示当前 SqlDataReader 是否包含一行或多行
	IsClosed	Public	只读属性，表示当前 SqlDataReader 实例是否已经关闭
	VisibleFieldCount	Public	只读属性，表示当前 SqlDataReader 中未隐藏的字段的数目
方法	Read	Public	使当前 SqlDataReader 前进到下一条记录，即：向前读取
	NextResult	Public	当读取批处理 Transact-SQL 语句的结果时，使数据读取器前进到下一个结果，注意：并非下一行记录
	IsDBNull	Public	确定指定列中是否包含不存在的或缺少的值，这只有在列数据允许为空时才可能为 true
	Close	Public	关闭当前 SqlDataReader 实例，关闭后，IsClosed 属性为 true
	GetFieldType	Public	获取指定列的数据类型的 Type
	GetDataTypeName	Public	获取源数据类型的名称
	GetName	Public	获取指定列的名称
	GetOrdinal	Public	在给定列名称的情况下获取列序号
	GetBoolean	Public	按照布尔 (bool) 类型读取指定列的数据
	GetByte	Public	按照字节 (byte) 类型读取指定列的数据
	GetBytes	Public	从指定的列偏移量将字节流读入缓冲区，并将其作为从给定的缓冲区偏移量开始的数组

续表

分 类	名 称	访 问 性	说 明
方法	GetChar	Public	按照字符(char)类型读取指定列的数据
	GetChars	Public	从指定的列偏移量将字符流作为数组从给定的缓冲区偏移量开始读入缓冲区
	GetDateTime	Public	按照日期时间(DateTime)类型读取指定列的数据
	GetDecimal	Public	按照Decimal类型读取指定列的数据
	GetDouble	Public	按照双精度浮点数(double)类型读取指定列的数据
	GetFloat	Public	按照单精度浮点数(float)类型读取指定列的数据
	GetInt16	Public	按照16位整数(short)类型读取指定列的数据
	GetInt32	Public	按照32位整数(int)类型读取指定列的数据
	GetInt64	Public	按照64位整数(long)类型读取指定列的数据
	GetString	Public	按照字符串(string)类型读取指定列的数据

从表 24.6 中可以看出, SqlDataReader 类为数据记录的读取提供了良好的支持。除了表中的函数外, SqlDataReader 类还提供将数据作为 SQL 数据类型读取, 作为 XML 数据读取等接口, 这些内容本章暂不做介绍, 读者可以查阅 MSDN 或相关书籍。

下面的代码演示将如何通过 SqlDataReader 读取不同类型的数据。其中, 方法 UseSqlDataReader()通过 SqlDataReader 类的 GetName()方法获取列名, 并打印。然后通过 Read()方法依次向前读取数据记录, 直到全部读取完成。最后通过 GetString()方法读取字符串类型的文本字段的值, 例如, 学号、性别等字段。通过 GetInt32()方法读取整数字段的值, 如年龄字段(使用 GetInt32()方法读取年龄字段, 需修改原表的定义, 将 age 字段设置为 int 类型)。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace _24._3._4
{
    class Program
    {
        static void Main(string[] args)
        {
            UseSqlDataReader();
        }

        static void UseSqlDataReader()
        {
            //数据库连接字符串
            string conStr = "Data Source=.\SQLEXPRESS;Initial
Catalog=mydb;Integrated Security=True";
            //用连接字符串创建数据库连接对象
            SqlConnection con = new SqlConnection(conStr);
            //创建 SQL 命令对象, 并设置它的数据库连接
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = con;
            //通过 Open() 连接到数据库
```



```


con.Open( );
//查询数据表中所有的数据记录,并打印
cmd.CommandText = "SELECT uid,name, major, sex,age FROM
Student";
SqlDataReader dr = cmd.ExecuteReader( );
System.Console.WriteLine("\nStudent 表的数据记录");
//依次打印数据表的列名
for (int colindex = 0; colindex < dr.FieldCount; colindex++)
{
    System.Console.Write("    {0}", dr.GetName(colindex));
}
System.Console.WriteLine( );
//依次读出所有的数据记录,并显示到界面
while (dr.Read( ))
{
    System.Console.Write(" {0}", dr.GetString(0));
    System.Console.Write(" {0}", dr.GetString(1));
    System.Console.Write(" {0}", dr.GetString(2));
    System.Console.Write(" {0}", dr.GetString(3));
    System.Console.Write(" {0}", dr.GetInt32(4));
    System.Console.WriteLine( );
}
dr.Close( );
con.Close( );
}
}
}

```

运行代码,输出结果如下所示。

Student 表的数据记录

学号	姓名	专业	性别	年龄
0000000000	张三	计算机科学与技术	男	24
0000000001	李四	计算机科学与技术	男	22
0000000002	张芬	计算机科学与技术	女	25

 **注意:** DataReader 类通过 Read() 方法只能向前读取数据记录,而且不能修改数据记录。

另外,如果一个字段的数据为空值,即 IsDBNull() 返回 true 时,直接使用 GetString() 等方法会产生异常。所以在不确定是否记录为空值的情况下,通常先通过 IsDBNull() 方法进行判断,再处理。

24.3.5 用 SqlDataAdapter 获取数据库记录

24.3.3 节和 24.3.4 节中所介绍的数据库访问都是基于连接的。有些时候需要对大量数据进行处理,且处理时间较长,不宜在连接状态下操作,这时需要使用基于无连接的数据库访问。在 ADO.NET 中,通过 SqlDataAdapter 和 DataSet 联合使用实现基于无连接的数据库访问。

SqlDataAdapter 类作为本地 DataSet (或 DataTable) 与数据库服务器之间的连接器,它表示用于填充 DataSet 和更新 SQL Server 数据库的一组数据命令和一个数据库连接。表 24.7 列出了它的常用成员。

表 24.7 SqlDataAdapter 类常用成员

分 类	名 称	访 问 性	说 明
属性	AcceptChangesDuringFill	Public	读写属性，表示在任何 Fill 操作过程中，是否接受本地记录中已经存在的数据记录更改
	AcceptChangesDuringUpdate	Public	读写属性，表示在 Update 操作过程中，是否接受本地记录中已经存在的数据记录更改
	ContinueUpdateOnError	Public	读写属性，表示在行更新过程中遇到错误时是否继续更新下一条记录，还是产生异常（停止更新操作）
	UpdateBatchSize	Public	读写属性，表示每次到服务器的往返过程中处理的数据记录行数
	DeleteCommand	Public	读写属性，表示从数据库服务器删除记录所使用的 Transact-SQL 语句或存储过程
	InsertCommand	Public	读写属性，表示向数据库服务器添加记录所使用的 Transact-SQL 语句或存储过程
	SelectCommand	Public	读写属性，表示从数据库服务器获取记录所使用的 Transact-SQL 语句或存储过程
	UpdateCommand	Public	读写属性，表示更新数据库服务器记录所使用的 Transact-SQL 语句或存储过程
方法	Fill	Public	从数据库服务器获取数据，填充到本地的数据集（DataSet）或数据表（DataTable）
	FillSchema	Public	从数据库服务器获取数据架构，填充到本地的数据集（DataSet）或数据表（DataTable）
	Update	Public	将本地数据集或数据表中的数据记录更改更新到数据库服务器，为 DataSet 中每个已插入、已更新或已删除的行调用相应的 INSERT、UPDATE 或 DELETE 语句

从表 24.7 中可以看出，DataAdapter 类的常用成员并不多，使用也非常简单，本节先介绍如何从数据库获取数据记录。使用 DataAdapter 类从数据库服务器获取数据记录到本地通常需要以下几个步骤。

（1）通过 DataAdapter 类构造函数创建一个可用的 DataAdapter 对象，同时为它指定数据库连接、查询命令等基本参数。DataAdapter 构造函数包括以下常用重载版本。

- ❑ `SqlDataAdapter()`: 创建一个默认的 SqlDataAdapter 对象，它的任何参数都在后期指定。
- ❑ `SqlDataAdapter(SqlCommand cmd)`: 创建一个具有指定查询命令的 SqlDataAdapter 对象，参数 cmd 表示用于从数据库获取数据的 SQL 命令。
- ❑ `SqlDataAdapter(string selectcmd, SqlConnection con)`: 创建一个具有指定查询命令和数据库连接的 SqlDataAdapter 对象，其中 selectcmd 表示查询命令的 SQL 语句，con 表示可用的数据库连接。
- ❑ `SqlDataAdapter(string selectcmd, string constr)`: 创建一个具有指定查询命令和数据库连接的 SqlDataAdapter 对象，其中 selectcmd 表示查询命令的 SQL 语句，constr 表示数据库连接的连接字符串。

（2）通过 SqlDataAdapter 类的 SelectCommand 属性设置或修改查询命令。如果不需要

修改, 则不需要此操作。

(3) 通过 `SqlDataAdapter` 类的 `FillSchema()` 方法从数据库服务器获取数据架构到本地数据集或数据表。如果只是需要数据结构, 则必须这样。

(4) 通过 `SqlDataAdapter` 类的 `Fill()` 方法从数据库服务器获取数据到本地数据集或数据表。

下面的代码将演示 `SqlDataAdapter` 获取数据记录的具体实现。`UseDataAdapterRead()` 方法首先创建一个 `SqlDataAdapter` 对象, 并通过构造函数指定它的查询 SQL 命令。然后依次通过 `FillSchema()` 和 `Fill()` 方法从数据库服务器获取数据, 最后通过 `DataTableReader` 方法读取, 并打印读取到的数据。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace _24._3._5
{
    class Program
    {
        static void Main(string[] args)
        {
            UseDataAdapterRead();
        }

        static void UseDataAdapterRead()
        {
            //数据库连接字符串
            string conStr = "Data Source=.\SQLEXPRESS;Initial Catalog=mydb;Integrated Security=True";
            //用连接字符串创建数据库连接对象
            SqlConnection con = new SqlConnection(conStr);
            //指定查询操作的命令
            string selstr = "SELECT uid, name, major, sex, age FROM Student";
            SqlDataAdapter da = new SqlDataAdapter(selstr, con);
            DataSet ds = new DataSet();
            //填充数据结构
            da.FillSchema(ds, SchemaType.Mapped);
            //从数据库用 Selecte 语句获取数据
            da.Fill(ds);
            //通过数据表创建一个本地的数据读取器
            DataTableReader dr = ds.CreateDataReader();
            //依次打印出所有的数据记录
            System.Console.WriteLine("数据记录如下:");
            while (dr.Read())
            {
                for (int index = 0; index < dr.FieldCount; index++)
                {
                    System.Console.Write(" {0}", dr[index].ToString());
                }
                System.Console.WriteLine();
            }

            //将本地的数据记录更改更新到数据库服务器
            da.Update(ds);
        }
    }
}
```

```

        con.Close();
    }
}

```

运行代码，打印出 Student 表的数据记录，如下所示。这里的输出根据数据库中实际的记录数会有所变化。

数据记录如下：

0000000000	张三	计算机科学与技术	男	24
0000000001	李四	计算机科学与技术	男	22
0000000002	张芬	计算机科学与技术	女	25

24.3.6 用 SqlDataAdapter 更改数据库记录

基于无连接的网络数据库同样可以修改数据库记录，只是修改的数据记录不会立刻更新到服务器，根据程序具体实现会出现不同程度的延时。通过 SqlDataAdapter 实现基于无连接的数据库访问通常需要以下几个步骤。

(1) 通过 DataAdapter 类构造函数创建一个可用的 DataAdapter 对象，同时为它指定数据库连接、查询命令等基本参数。

(2) 创建一个和 DataAdapter 类相关的 SQL 命令创建器 (SqlCommandBuilder) 对象，并通过 SqlCommandBuilder 类的 GetInsertCommand()、GetUpdateCommand()、GetDeleteCommand() 方法获取与 DataAdapter 类的 SQL 命令对应的删除、修改、添加命令。

(3) 通过 DataAdapter 类的 FillScheme() 或 Fill() 方法从数据库服务器获取数据记录到本地数据集 DataSet 或 DataTable 中。

(4) 通过 DataSet 或 DataTable 的属性和方法等方式（见 24.2 节）添加、删除、修改本地的数据记录。

(5) 通过 DataAdapter 类的 Update() 方法将本地数据记录的修改提交到数据库服务器。其中 DataAdapter 类的 Update() 方法具有多个重载版本，可以根据需要提交指定数据，它的常用重载定义如下所示。

- ❑ int Update (DataRow[] rows)：只提交指定行的数据到服务器，其中，参数 rows 表示要提交的数据记录的数组。
- ❑ int Update (DataSet ds)：提交指定数据集中所有被更改的数据记录，其中，参数 ds 表示要提交的 DataSet。
- ❑ int Update (DataTable dt)：提交指定数据表中所有被更改的数据记录，其中，参数 dt 表示要提交的 DataTable。

下面的代码演示如何通过 SqlDataAdapter 类完成数据库数据的更改。其中，AddStudent() 方法将学生信息添加到本地数据表中。UseDataAdapterUpdate() 方法则通过上面提到的 5 个步骤添加几个学生信息到本地数据表，然后通过 SqlDataAdapter 类的 Update() 方法提交到数据库服务器。最后通过 UseDataAdapterRead() 方法将新的数据表中新的数据记录打印出来。

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Data;

```



```

using System.Data.SqlClient;

namespace _24._3._6
{
    class Program
    {
        static void Main(string[] args)
        {
            UseDataAdapterUpdate( );
            UseDataAdapterRead( );
        }

        static void UseDataAdapterRead( )
        {
            //数据库连接字符串
            string conStr = "Data Source=.\SQLEXPRESS;Initial
            Catalog=mydb;Integrated Security=True";
            //用连接字符串创建数据库连接对象
            SqlConnection con = new SqlConnection(conStr);
            //指定查询操作的命令
            string selstr = "SELECT uid, name, major, sex, age FROM Student";
            SqlDataAdapter da = new SqlDataAdapter(selstr, con);
            DataSet ds = new DataSet();
            //填充数据结构
            da.FillSchema(ds, SchemaType.Mapped);
            //从数据库用 Selecte 语句获取数据
            da.Fill(ds);
            //通过数据表创建一个本地的数据读取器
            DataTableReader dr = ds.CreateDataReader( );
            //依次打印出所有的数据记录
            System.Console.WriteLine("数据记录如下:");
            while (dr.Read( ))
            {
                for (int index = 0; index < dr.FieldCount; index++)
                {
                    System.Console.Write(" {0}", dr[index].ToString( ));
                }
                System.Console.WriteLine( );
            }

            //将本地的数据记录更改更新到数据库服务器
            da.Update(ds);
            con.Close( );
        }

        static void UseDataAdapterUpdate( )
        {
            //数据库连接字符串
            string conStr = "Data Source=.\SQLEXPRESS;Initial
            Catalog=mydb;Integrated Security=True";
            //用连接字符串创建数据库连接对象
            SqlConnection con = new SqlConnection(conStr);
            //指定查询操作的命令
            string selstr = "SELECT uid, name, major, sex, age FROM Student";
            SqlDataAdapter da = new SqlDataAdapter(selstr, con);
            //指定 SQL 命令构建器--SqlCommandBuilder 对象
            SqlCommandBuilder cmdBuilder = new SqlCommandBuilder(da);
            //生成 DataAdapter 的删除、更改、添加命令

```



```

        da.DeleteCommand = cmdBuilder.GetDeleteCommand( );
        da.UpdateCommand = cmdBuilder.GetUpdateCommand( );
        da.InsertCommand = cmdBuilder.GetInsertCommand( );
        //新建数据集合
        DataSet ds = new DataSet( );
        //填充数据结构
        da.FillSchema(ds, SchemaType.Mapped);
        DataTable dt = ds.Tables[0];
        //添加学生信息
        AddStudent(dt, "张三", "0000000101", "男", "计算机科学与技术", 20);
        AddStudent(dt, "李四", "0000000103", "男", "计算机科学与技术", 25);
        AddStudent(dt, "王霞", "0000000201", "女", "中国古典文学", 22);
        //将本地的数据记录更改更新到数据库服务器
        da.Update(ds);
        con.Close( );
    }

    static void AddStudent(DataTable dt, string name, string id,
        string xingbie, string major, int age)
    {
        //根据表的模式新建数据记录行
        DataRow row = dt.NewRow( );
        //设置数据记录的数据
        row["UID"] = id;
        row["Name"] = name;
        row["Major"] = major;
        row["Sex"] = xingbie;
        row["Age"] = age;
        //将数据记录添加到数据表
        dt.Rows.Add(row);
    }
}

```

运行程序，得到输出结果如下，在运行前保证这些学生的信息在数据库中不存在。从中可以看出，新增加的学生信息被成功提交到数据库服务器中。

数据记录如下：

0000000101	张三	计算机科学与技术	男	20
0000000103	李四	计算机科学与技术	男	25
0000000201	王霞	中国古典文学	女	22

24.4 用 ADO.NET 访问 Access 数据库

Access 作为微软 Office 办公软件的主要组件，是 Windows 系统下常用的本地数据库，在很多中小应用软件中用作保存本地数据。本节将介绍如何通过 ADO.NET 访问 Access 数据库。

24.4.1 System.Data.OleDb 命名空间提供的功能

在 ADO.NET 中，通过 OleDb 数据源驱动程序访问 Access 数据库，它通过

System.Data.OleDb 命名空间提供的类和结构实现。System.Data.OleDb 命名空间下的主要结构与 System.Data.SqlClient 命名空间非常相似，同样包括以下几个主要功能的类。

- ❑ OleDbConnection: Ole 数据库连接类，通过它可以连接到指定的 Ole 数据库（如 Access 数据库），并且提供打开和关闭数据库连接功能。
- ❑ OleDbCommand: Ole 数据库命令类，它表示一个可执行的 SQL 命令，可以是普通的文本 SQL 命令，也可以是带参数的 SQL 命令。SQL 命令可以通过 OleDbCommandBuilder 类创建。
- ❑ OleDbDataAdapter: Ole 数据适配器类，用于连接 SQL Server 数据库与本地 DataSet 和 DataTable 数据集，提供获取服务器数据、更新本地更改等操作。
- ❑ OleDbDataReader: Ole 数据读取器，提供只读只向前的数据记录读取。

OleDb 数据库的访问和 SQL Server 数据库访问一样，首先通过 OleDbConnection 连接到数据库，然后通过 OleDbCommand、OleDbDataReader 类实现基于连接的数据库访问。也可以通过 OleDbCommand、OleDbDataAdapter、DataSet、DataTable 类实现基于无连接的数据库访问。

24.4.2 访问 Access 数据库的各种类

Access 数据库的访问与 SQL Server 数据库访问相似，都可以分成连接和无连接两种访问模式。只是 Access 数据库访问是使用 OleDb 数据源驱动程序通过 System.Data.OleDb 中的类实现，这几个重要类具有一对一的关系，如表 24.8 所示。

表 24.8 OleDb与SqlClient主要类对比

OleDb空间类	SqlClient空间类	功能说明
OleDbConnection	SqlConnection	它们都从DbConnection类派生而来，表示一个数据库连接。通过指定的连接字符串连接到数据库，不同数据库具有不同的连接属性。例如，SQL Server需要指定数据库服务器地址、目标数据库、安全性、用户名和密码等。Access数据库需要指定数据库文件（.mdb文件）、用户名和密码等
OleDbCommand	SqlCommand	它们都从DbCommand类派生而来，表示一个数据库SQL命令类。实现基于不同数据库的SQL查询语句，包括Select语句、Insert语句、Update语句、Delete语句等。根据不同的数据库类型这些语句字符串具有一定的区别
OleDbDataReader	SqlDataReader	它们都从DbDataReader类派生而来，表示一个数据读取器。实现一个只读只向前的数据库记录读取器，实现基于有连接的数据库访问
OleDbDataAdapter	SqlDataAdapter	它们都从DbDataAdapter类派生而来，表示一个数据连接类。实现数据库服务器到本地数据记录之间的连接，实现基于无连接的数据库访问

从表 24.8 中可以看出，Access 数据库的访问和 SQL Server 数据库的访问所采用的主要数据库访问类都是从相同的基类（如 DbConnection、DbCommand 等）派生而来。所以，在功能和使用方式上都完全相同。这里就不再重复介绍 Access 数据库的操作细节，读者只需要参照 24.3 节中关于 SQL Server 数据库访问的实例和讲解即可。

24.5 使用数据库访问控件

对数据库中的数据进行添加、查看、修改等操作通常需要通过 Windows 窗体界面来完成。在 .NET 类库中提供几个与此相关的 Windows 窗体控件，可以用来显示、浏览、修改数据记录，本节将介绍这些控件的使用方法。

24.5.1 用 DataGridView 控件管理数据库中的记录

在 .NET 类库中，DataGridView 控件提供一种强大而灵活的以表格形式显示数据的方式。可以使用 DataGridView 控件显示少量数据的只读视图，也可以对其进行缩放以显示特大数据集的可编辑视图。

可以用多种方式扩展 DataGridView 控件，将自定义行为内置在应用程序中。例如，可以通过编程方式指定自己的排序算法，创建自己的单元格类型。通过选择一些属性，可以轻松自定义 DataGridView 控件的外观。可以将许多类型的数据存储区用做数据源，也可以在未绑定数据源的情况下操作 DataGridView 控件。

在本节的实例中，将为读者演示 DataGridView 控件的使用。该实例使用的是一个 Access 数据，可将本章前面使用的 SQL Server 数据库 mydb 导出为一个 Access 数据库 student.mdb。完成本实例大致包括以下几个步骤：

(1) 新建 Windows 窗体应用程序，并命名为 UseDbControls，在“窗体设计器”中打开窗体，并从“工具箱”控件窗体中将 DataGridView 控件（在“数据”分组中）拖放到窗体上。

(2) 选中 DataGridView 控件，并在“属性管理器”中打开，然后编辑它的属性，DataSource 属性项显示为“无”，此时 DataGridView 控件只是一个空白，没有绑定任何数据源。

(3) 通过 DataSource 属性的下拉框，选择“添加项目数据源”得到“数据源配置向导”界面，选择“数据库”选项作为数据源，然后单击“下一步”按钮进入数据库模型设置界面，如图 24.1 所示。



图 24.1 选择数据库模型

(4) 在图 24.1 所示界面中单击“数据集”选项，再单击“下一步”按钮进入“数据源配置向导”对话框，如图 24.2 所示。可以通过“新建连接”按钮创建一个新的数据库连接，也可以直接从下拉列表中选择项目中已有的数据库连接。

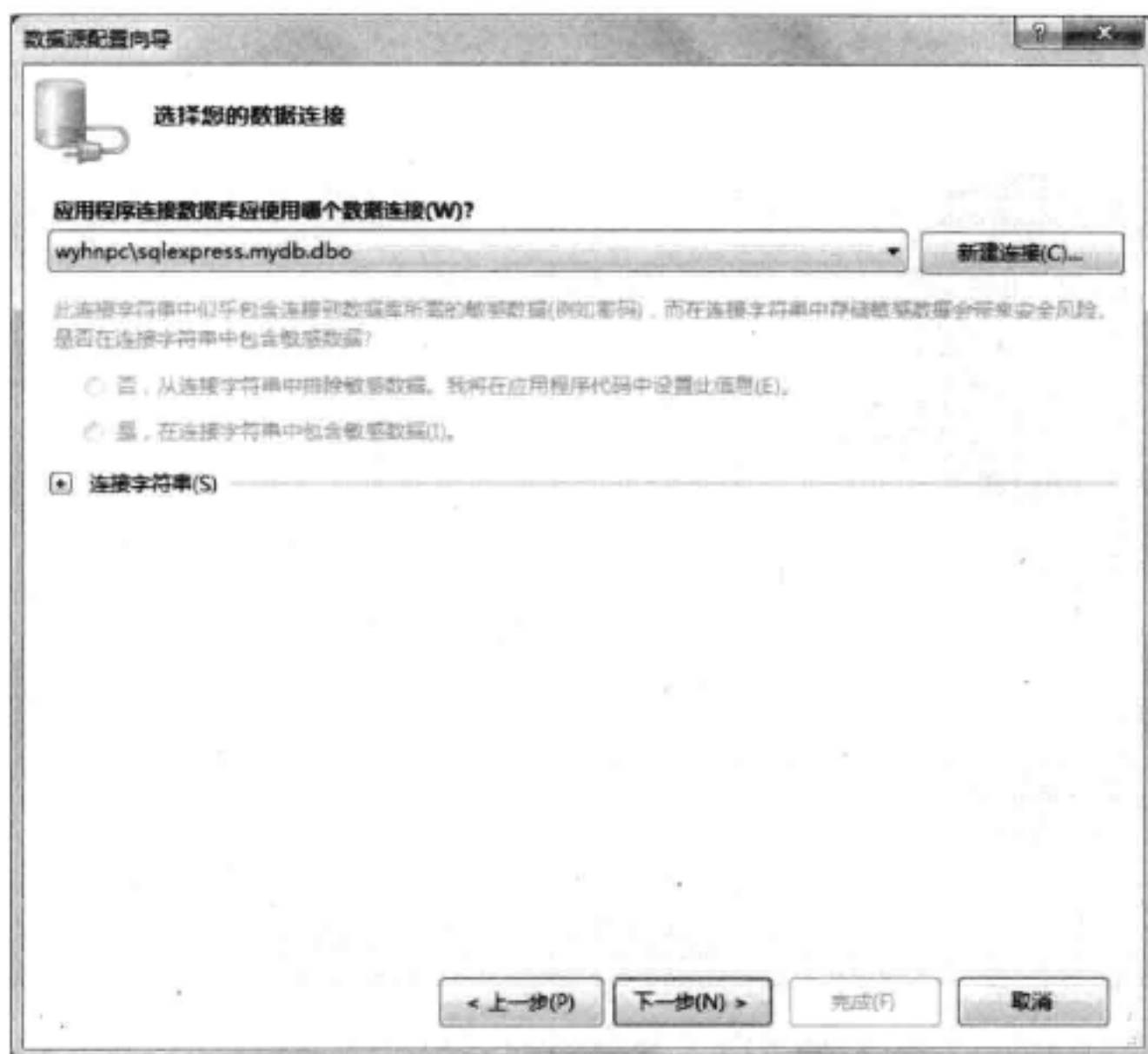


图 24.2 选择数据连接

(5) 单击“新建连接”按钮打开如图 24.3 所示“选择数据源”窗口，选择“Microsoft Access 数据库文件”选项。

(6) 单击“继续”按钮，打开如图 24.4 所示的“添加连接”窗口选择数据库 student.mdb。

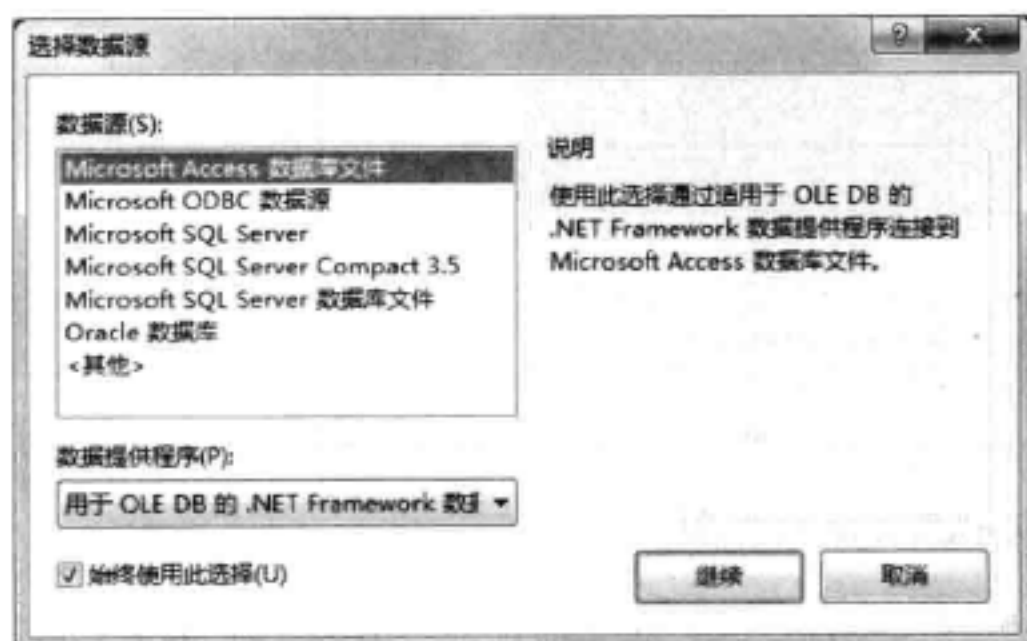


图 24.3 选择数据源



图 24.4 添加连接

(7) 单击“确定”按钮返回图 24.2 所示的界面，只是在连接下拉列表框中显示为 student.mdb。

(8) 单击“下一步”按钮保存连接字符串,再单击“下一步”按钮选择要连接的数据表或视图,也可以指定要显示的字段,如图 24.5 所示。



图 24.5 选择数据表和字段

(9) 这样, DataGridView 控件在窗体设计器中就会显示出数据记录的列结构,如图 24.6 所示,其中列信息为 UID、Name、Sex、Age 和 Major 都是根据第(8)步的配置从数据库中获取而来。

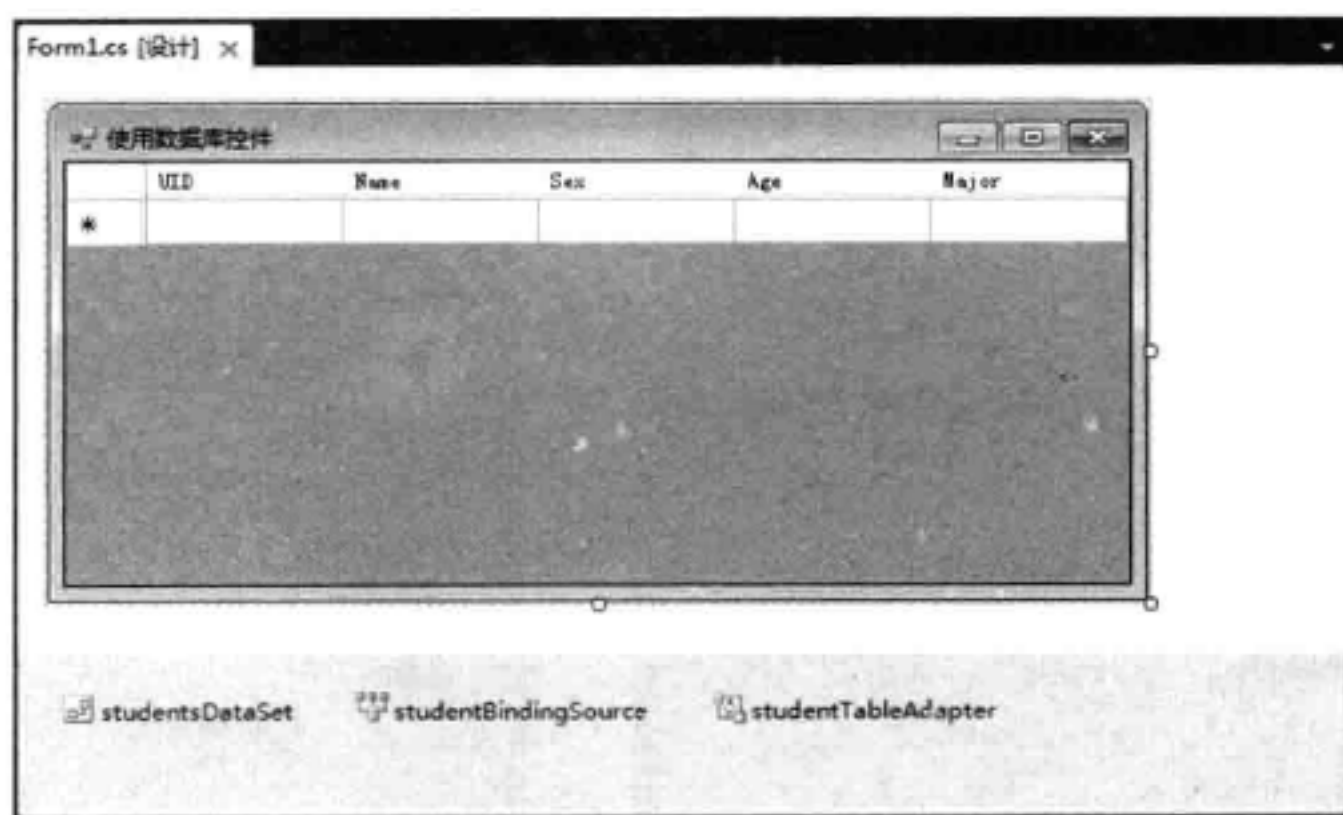


图 24.6 DataGridView 设计图

从图 24.6 中还可以看到, Visual Studio 2010 自动生成 3 个不可视控件,即 studentsDataSet、studentsBindingSource 和 studentTableAdapter。其中, studentsDataSet 表示数据库 Student 被应用程序访问的本地数据集(DataSet)。studentTableAdapter 则表示被访问的数据表的 DataAdapter 对象,它的类型会根据数据库类型自动调整,可能是 SqlDataAdapter,也可能是 OleDbDataAdapter。

完成上面的操作后, Visual Studio 2010 为开发人员自动生成用于无连接的数据库访问的成员, 并在窗体的 Load 事件处理函数中自动添加数据的获取代码。下面代码是 Visual Studio 2010 自动生成的填充数据表的代码。

```
namespace UseDbControls
{
    public partial class FrmUseDbControl : Form
    {
        public FrmUseDbControl()
        {
            InitializeComponent();
        }

        private void FrmUseDbControl_Load(object sender, EventArgs e)
        {
            // TODO: 这行代码将数据加载到表“studentsDataSet1.Student”中。可以
            //根据需要移动或删除它
            this.studentTableAdapter.Fill(this.studentsDataSet.Student);
        }
    }
}
```

至此, 用户可以通过 DataGridView 控件完成包括添加、修改和删除 3 种操作的数据库操作, 但是这些操作都是对本地数据库的更改。DataGridView 控件不会自动将这些更改提交到数据库服务器, 通过 DataGridView 类的 RowStateChanged 事件可以监视 DataGridView 控件中数据记录的修改。可以在 DataGridView 类的 RowStateChaged 事件处理函数或其他函数中将本地更改提交到数据库服务器。例如下面代码, 就是通过 DataGridView 类的 RowStateChanged 事件将本地更改提交到数据库服务器中。

```
private void dgvUserInfo_RowStateChanged(object sender,
    DataGridViewRowStateChangedEventArgs e)
{
    this.studentTableAdapter.Update(this.studentsDataSet);
}
```

从其中可以看出, 数据记录的提交实际上也是通过 DbDataAdapter 类的 Update() 方法来实现, DataGridView 本身并没有这些功能。

DataGridView 控件除了显示和管理数据库记录之外, 还可以用简单的表格形式来管理本地内存或文件中的数据。另外, DataGridView 控件还提供很多属性管理控件的使用, 比如, 可以设置它的背景色、前景色、字体的属性, 还可以分别设置列标题和行标题的样式, 可以设置是否记录、是否可以编辑等。DataGridView 控件的功能十分强大, 有兴趣的读者可以查阅 MSDN 或相关书籍, 进一步学习它的高级使用方法。

24.5.2 用 BindingNavigator 控件导航记录

24.5.1 节介绍的 DataGridView 控件可以显示、编辑数据表中所有的数据, 但是在删除等操作时只能通过快捷键方式完成, 这样并不方便。在 .NET 类库中, 还提供了另外一个控件——BindingNavigator, 它为对绑定数据的导航和操作提供快捷工具栏。

在实例 UseDbControls 中, 为 DataGridView 控件添加一个 BindingNavigator 控件协助完成数据记录的编辑和导航, 需要以下几个步骤来完成。

(1) 在“窗体设计器”中打开主界面, 从“工具箱”控件窗体中将一个 BindingNavigator 控件拖到界面上, 并设置它的 Dock 属性为 Top, 如图 24.7 所示。

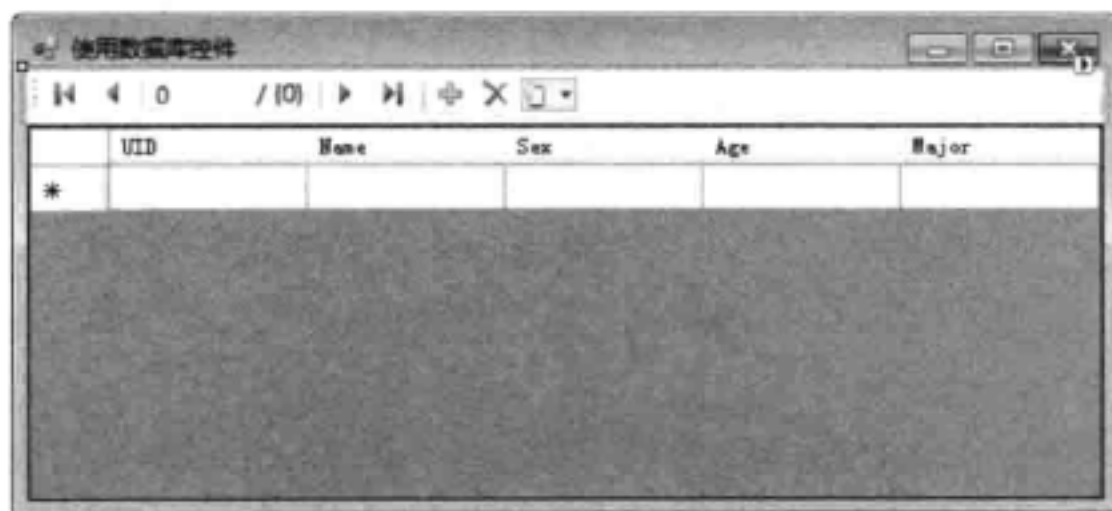


图 24.7 BindingNavigator 设计图

(2) 在“属性管理器”中选择并设置 BindingNavigator 控件的 DataSource 属性与 DataGridView 控件一样都为 studentBindingSource。这样, 它们都绑定到同一个数据源了。

(3) 生成并运行程序, 可以看到 BindingNavigator 控件对 DataGridView 控件的导航效果, 还可以通过“添加”和“删除”按钮直接从 DataGridView 控件中进行对应操作。

图 24.7 中的 BindingNavigator 控件所包含的按钮是 .NET 类库自带的, 开发人员可以自行设计对应功能的按钮, 也可以新增其他功能的按钮。

24.6 本章总结

ADO.NET 是一组向 .NET 程序员公开数据访问服务的类。ADO.NET 为创建分布式数据共享应用程序提供了一组丰富的组件, 它对 Microsoft SQL Server 和 XML 等数据源, 以及通过 OLE DB 和 XML 公开的数据源提供一致的访问。数据共享使用者应用程序可以使用 ADO.NET 连接到这些数据源, 并检索、处理和更新所包含的数据。

ADO.NET 通过数据处理将数据访问分解为多个可以单独使用或一前一后使用的不连续组件。ADO.NET 包含用于连接到数据库、执行命令和检索结果的 .NET Framework 数据提供程序。可以直接处理检索到的结果, 或将其放入 ADO.NET DataSet 对象, 以便与来自多个源的数据或在层之间进行远程处理的数据组合在一起, 以特殊方式向用户公开。ADO.NET DataSet 对象也可以独立于 .NET Framework 数据提供程序使用, 以管理应用程序本地的数据或源自 XML 的数据。

24.7 实战练习

1. 在 Visual Studio 2010 中新建一个控制台应用程序, 编写程序将 SQL Server 数据库 mydb 中 Student 表的所有数据输出到控制台。
2. 在 Visual Studio 2010 中新建一个控制台应用程序, 编写程序将 SQL Server 数据库

mydb 中 Student 中凡是年龄小于 23 岁的增加 1 岁，将修改后的数据保存到数据，然后再显示修改后的结果到控制台。

3. 在 Visual Studio 2010 中新建一个控制台应用程序，编写程序将 SQL Server 数据库 mydb 中 Student 表和 Major 表添加到 DataSet 中，并将这两个表的内容输出到控制台。

4. 在 Visual Studio 2010 中新建一个 Windows 窗体应用程序，在窗体中用 DataGridView 控件显示 SQL Server 数据库中 Student 表中的内容，并在 DataGridView 控件上方添加一个 BindingNavigator 控件，用来控制数据的添加、修改和删除操作。

第 25 章 ASP.NET 技术入门

随着 Internet 的快速发展,网络在人们的日常生活中占据了越来越重要的地位。各式各样的网站也成了一种主要的信息来源,随着 Web 2.0 概念的提出,以及 XML 标准的普及,网站开发逐步发展成软件开发的一个重要部分。本章将简要介绍如何在 .NET4.0 下使用 C# 开发网页。

25.1 初识 ASP.NET

早期的网页开发语言包括 ASP、JSP、PHP 等,其中 ASP 是由微软推出的主流网页开发语言。但是随着 .NET 平台的推出,ASP 逐渐被 ASP.NET 所取代。本节将简要介绍 ASP.NET 的基本概念。

25.1.1 了解 ASP.NET

随着 .NET 框架的推出,微软对原有的网络开发语言 ASP 进行改革,推出性能更好,使用更加方便的网页开发语言 ASP.NET。ASP.NET 并不是 ASP 简单的版本改进,它和 ASP 有着完全不同的结构和开发模式,ASP.NET 使得开发网页可以更加方便和快捷。

ASP.NET 作为 .NET 框架的一个独立组件存在,提供一个统一的 Web 开发模型。ASP.NET 建立在公共语言运行库上的编程框架,包括 Web 应用程序开发所必需的各种服务,可用于在服务器上生成功能强大的 Web 应用程序。另外,ASP.NET 与 .NET 框架无缝集成,在开发 Web 应用程序时,可以访问 .NET 类库中的类,也可以访问使用 C#、VB.NET 等语言开发的自定义类库(DLL 文件)。在开发过程中使用 C#、VB.NET 等开发语言,可以充分利用公共语言运行库、类型安全、继承等多种语言特性。

与以前的 Web 开发模型(比如 ASP)相比,ASP.NET 提供了以下几个重要的优点。

- ❑ 增强的性能: ASP.NET 是在服务器上运行的编译好的公共语言运行库代码。与早期的 Web 开发模型不同,ASP.NET 可利用早期绑定、实时编译、本机优化等技术,相当于在编写代码行之前便显著提高了性能。
- ❑ 开发工具: Visual Studio 2010 为 ASP.NET 的开发提供了工具箱和设计器,可以通过控件拖放、属性设置等简单方式进行界面设计,后台通过 C#、VB.NET 等代码编辑器编码,使得开发 Web 应用程序更加方便快捷。
- ❑ 灵活性: 由于 ASP.NET 和 .NET 框架集成,因此 Web 应用程序开发人员可以利用整个 .NET 框架丰富的类库、消息处理和数据访问等技术。而且 ASP.NET 与语言无关,所以可以选择最适合应用程序、程序员最熟练的语言,或跨多种语言开发

Web 应用程序。

- ❑ 扩展性: ASP.NET 中, 可以编写自定义组件扩展或替换 ASP.NET 运行库的任何子组件。

25.1.2 System.Web 常用的类

和 Windows 窗体应用程序一样, 在 .NET 类库中也为 Web 应用程序提供了大量的支持数据类型, 它们都封装在 .NET 类库的 System.Web 命名空间下。System.Web 命名空间提供可以进行浏览器与服务器通信的类和接口, 主要包括以下几类。

- ❑ HttpRequest 类: 用于提供有关当前 HTTP 请求的广泛信息。
- ❑ HttpResponse 类: 用于管理对客户端的 HTTP 输出。
- ❑ HttpServerUtility 类: 用于提供对服务器端实用工具与进程的访问。
- ❑ 其他: 还包括用于 Cookie 操作、文件传输、异常信息和输出缓存控制的类。

通过这些类可以对整个网页的具体信息、当前连接信息、浏览器的 Cookie 情况进行管理, 还可以控制页面的缓存等。

除了以上这些和数据处理相关的类, Web 应用程序开发还需要一个重要的命名空间: System.Web.UI, 它提供的类和接口可以用来创建在网页中作为用户界面元素的 ASP.NET 服务器控件和网页。System.Web.UI 命名空间的结构和 System.Windows.Forms 命名空间类似, 也包括 Control 类, 为所有服务器控件提供一组通用功能。另外, System.Web.UI 命名空间还包括 Page 类 (类似于 Windows 窗体中的 Form 类), 每次对包含在 Web 应用程序中的 aspx 文件发出请求时, 都会自动生成一个 Page 类的对象。从 Control 类和 Page 类继承, 可以得到任何使用在网页中的用户界面元素。

.NET 类库为 Web 开发人员提供常用的网页界面元素, 如按钮、文本框、菜单等, 它们都包含在 System.Web.UI.WebControls 命名空间下。该命名空间定义的类型用来创建运行在服务器上的窗体控件, 不仅包括按钮、文本框、菜单、列表等常用控件, 还包括日历等具有特殊用途的控件。

System.Web.UI.WebControls 命名空间下提供的 Web 服务器控件, 虽然最后都表现为 HTML 标记语言, 但由于它们运行在服务器上, 因此可以以编程方式控制这些元素。其中, WebControl 类用这些服务器控件类的基类为它们提供最基本的实现。这些常用的 Web 控件, 将在本章 25.2 节介绍, 更多相关的信息, 读者还需要自行查阅 MSDN。

25.1.3 创建一个 Web 应用程序

通过 Visual Studio 开发环境, 开发人员在开发 Web 应用程序时, 默认是以前后台模式完成的, 这种模式也是最简单方便的。所谓前后台模式是指在 Web 应用程序开发时, 前台的网页界面是一个以 aspx 为后缀的脚本文件, 该脚本文件被集成在 IIS 中的 ASP.NET 所编译, 然后被客户端访问。在 ASPX 文件后台, 还包含一个具体的代码文件, C# 语言则以 cs 为后缀, VB.NET 语言则以 vb 为后缀。

在前后台开发模式中, 前台的 aspx 文件只包含网页的界面部分, 包括控件布局、颜色、样式、主题等, 后台的代码文件 (如 cs 文件) 则负责逻辑功能的具体实现, 比如访问数据

库、数据运算等。前后台的开发模式的最大好处是界面和逻辑分离，界面设计人员和代码开发人员可以完全独立各自工作，不必受到彼此的约束。

在对 ASP.NET 有了基本了解之后，现在创建第一个 Web 应用程序，这是一个简单的网页，只有一个欢迎画面。需要以下 4 个步骤来完成。

(1) 打开 Visual Studio 2010 开发环境，通过选择“文件”|“新建”|“网站”命令，打开“新建网站”对话框，如图 25.1 所示。



图 25.1 新建网站

(2) 在左侧模板列表中选择 Visual C# 选项，接着在右侧模板列表中选择“ASP.NET 网站”选项，在“位置”下拉列表框中选择“文件系统”选项，并浏览要保存网站的本地目录。

(3) 单击“确定”按钮完成新网站的创建。

(4) 成功创建后，Visual Studio 会自动生成两个网页：Default.aspx 和 About.aspx，这两个网页文件还分别对应代码文件 Default.aspx.cs 和 About.aspx.cs，是这两个网页的后台实现代码。

如果在图 25.1 中选择“ASP.NET 空网站”选项，则不会创建上面提到的两个网页文件。

在“解决方案资源管理器”中双击 Default.aspx 可以在 Web 窗体设计器中打开该网页进行编辑，包含“设计”和“源”两种编辑模式。“设计”模式是所见即所得的编辑方式，可以直接将“工具箱”中的控件拖到设计器，可以通过“属性管理器”设置网页和控件的外观属性等。“源”模式则是直接查看 ASP.NET 语言代码，可以直接从网页的界面代码进行修改。

(5) 在这里，首先用“设计”模式进行编辑，可以看到如图 25.2 所示的内容。虽然新建网站后还未进行任何的设计，但在网页中可以看到有很多的内容，并具有配置好的颜色。这是由新建网站时系统自动生成的网站页面（包含母版页）。

(6) 在界面上修改内容，将“欢迎使用 ASP.NET!”的内容修改为“朋友，欢迎您的到来！”，并通过“工具栏”编辑它的颜色和字体等属性，然后删除下面的一些文字（由于最上方的文字“我的 ASP.NET 应用程序”属于母版页，所以不能直接修改这些内容）。

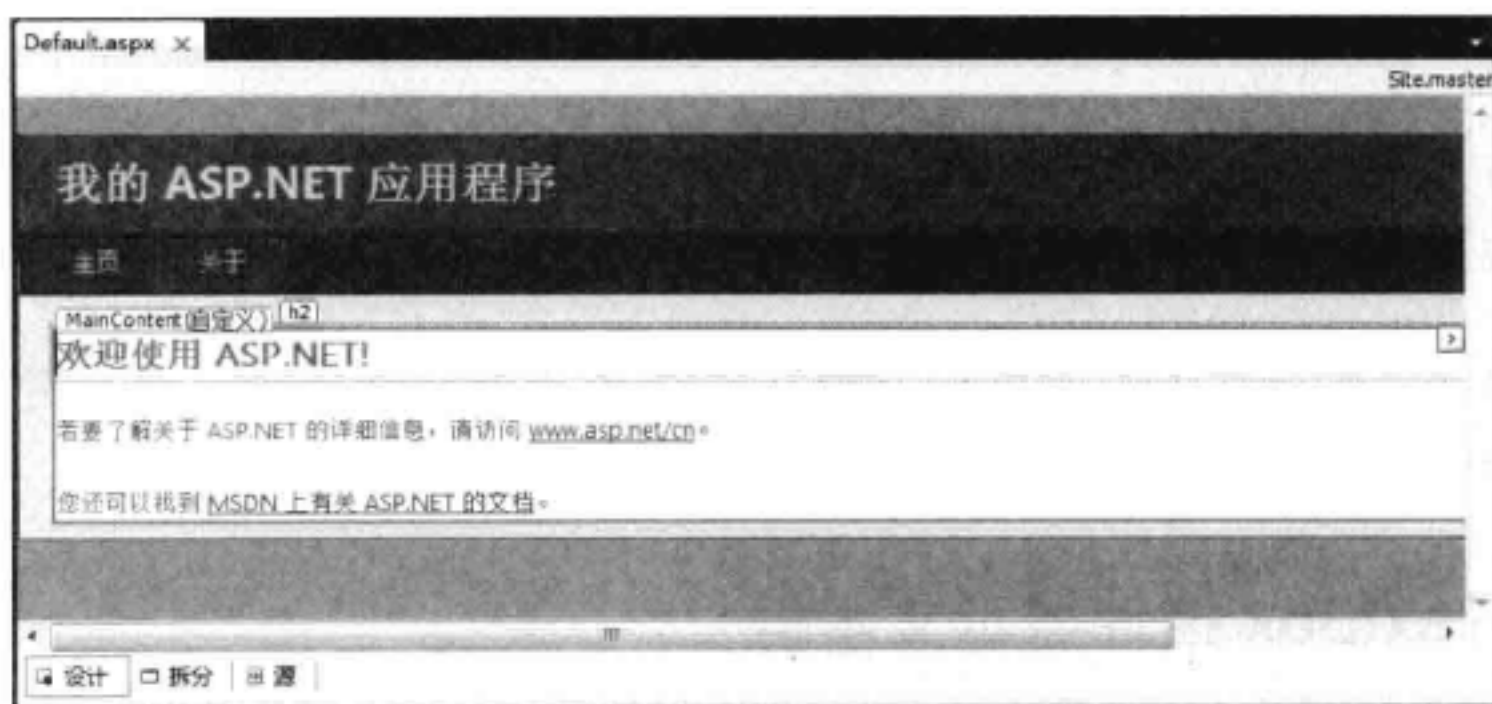


图 25.2 Default 页面

(7) 切换到“源”模式，将页面标题设置为“欢迎您的到来”。最后从“源”模式下看到的 ASP.NET 代码如下所示。

```
<%@ Page Title="欢迎您的到来" Language="C#" MasterPageFile="~/Site.master"
    AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<asp:Content ID="HeaderContent" runat="server"
    ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server"
    ContentPlaceHolderID="MainContent">
    <h2> 朋友，欢迎您的到来! </h2>
</asp:Content>
```

在前面的代码中，`<%@ Page...%>`节点是对整个页面属性的设置，这些属性可以通过“属性管理器”设置。这里，`Language`属性表示后台代码文件的语言，如 C#。`AutoEventWireup`属性表示控件和页面的事件是否自动匹配，通常为 `true`。`CodeFile`属性则表示后台代码文件名，如 `Default.aspx.cs`。`Title`则定义网页的头部信息，是网页的标题。在下面有两个内容区域，用来显示网页中的内容。如“`ContentPlaceHolderID="HeadContent"`”部分可放置一部分面容，而在当前网页中，在“`ContentPlaceHolderID="MainContent"`”部分用来放置实现网页呈现给用户的具体内容。

通过选择“调试”|“启动调试”命令可以查看网页的效果如图 25.3 所示。注意，在 Visual Studio 2010 中并不要求当前计算机中要安装 IIS。

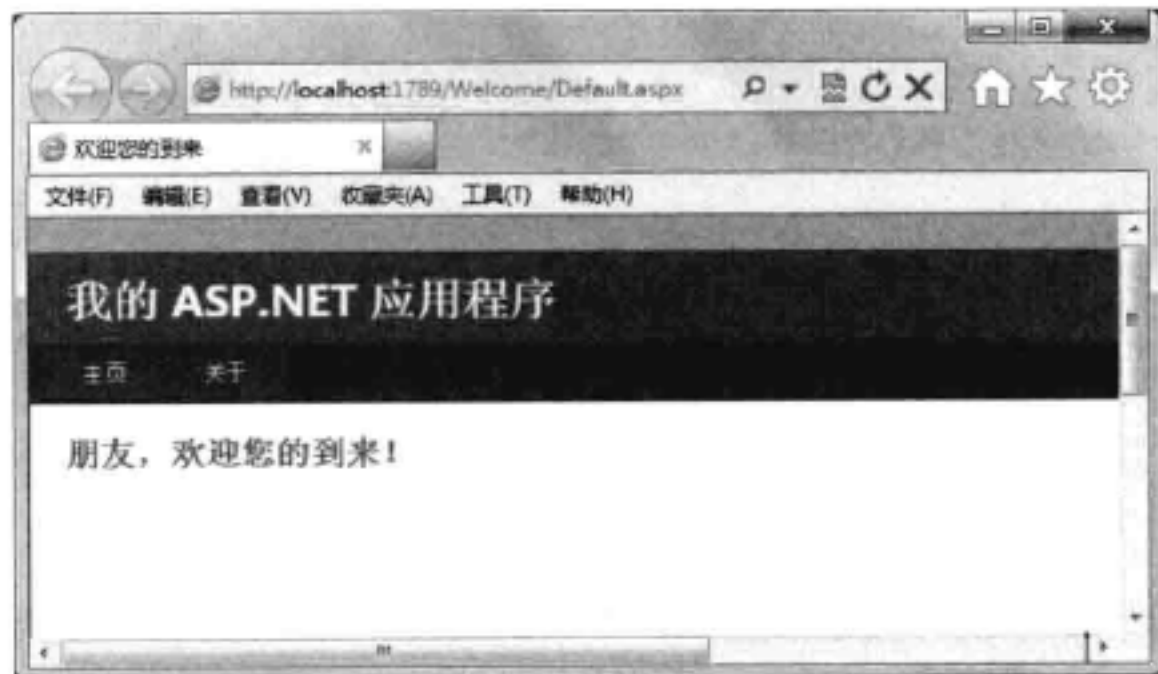


图 25.3 欢迎页面效果图

25.2 用 ASP.NET 控件创建网页

.NET 类库为网站开发人员提供了多种丰富的 Web 服务器控件，这些控件包括按钮、文本框、列表框等，本节将详细介绍其中最常用的几个 Web 服务器控件。

25.2.1 用 TextBox 控件显示文本框

在网页开发中，文本输入框控件是由 `System.Web.UI.TextBox` 类实现，为用户提供了一种向 Web 窗体页中键入信息（包括文本、数字和日期）的方法。TextBox 服务器控件由脚本 `<asp:TextBox>` 声明，属性 `runat` 的值为 `server`，表示该控件为服务器控件，以下代码表示控件 `TextBox1` 是服务器端运行的文本输入框控件。

```
<asp:TextBox ID="TextBox1" runat="server">默认文本框</asp:TextBox>
```

Web 窗体中的 TextBox 控件具有 3 种输入模式：单行、密码和多行。其中，单行模式只接收单行的文本输入，密码模式也只能接收单行文本输入，但是输入字符用“*”隐藏，多行模式则允许用户输入多行文本。另外，TextBox 文本输入框还有字体和颜色、边框和背景等属性可以设置，这些属性都可以通过“属性管理器”来完成。

下面代码中，定义了 4 个文本输入框，它们都是单行输入模式，TextBox1 只是简单的单行文本输入框，TextBox2 则是红色且字体为楷体的文本输入框；TextBox3 是修改了边框样式和背景颜色的输入框；TextBox4 是一个 `Enable` 属性为 `false` 的不可用文本框；TextBox5 则是一个只读文本框。

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>使用文本框</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      默认文本框: <asp:TextBox ID="TextBox1" runat="server">默认文本框
      </asp:TextBox>
      <br />
      红色楷体文本框: <asp:TextBox ID="TextBox2" runat="server" Font-Names="
      楷体_GB2312" ForeColor="Red">红色楷体文本</asp:TextBox>
      <br />
      边框修改文本框: <asp:TextBox ID="TextBox3" runat="server" BackColor=
      "#FFC0C0" BorderColor="#00C000" BorderStyle="Dashed" Font-Names="
      楷体_GB2312" ForeColor="Gray">边框修改文本框</asp:TextBox>
      <br />
      不可用文本框: <asp:TextBox ID="TextBox4" runat="server" Enabled=
      "False">不可用文本框</asp:TextBox>
      <br />
      只读文本框: <asp:TextBox ID="TextBox5" runat="server" ReadOnly=
      "True">只读文本</asp:TextBox>
    </div>
  </form>
```



```
</body>
</html>
```

上面代码得到各种文本框的效果如图 25.4 所示。文本框控件的属性还有很多，读者可以通过“属性管理器”修改不同的属性得到各种外观。

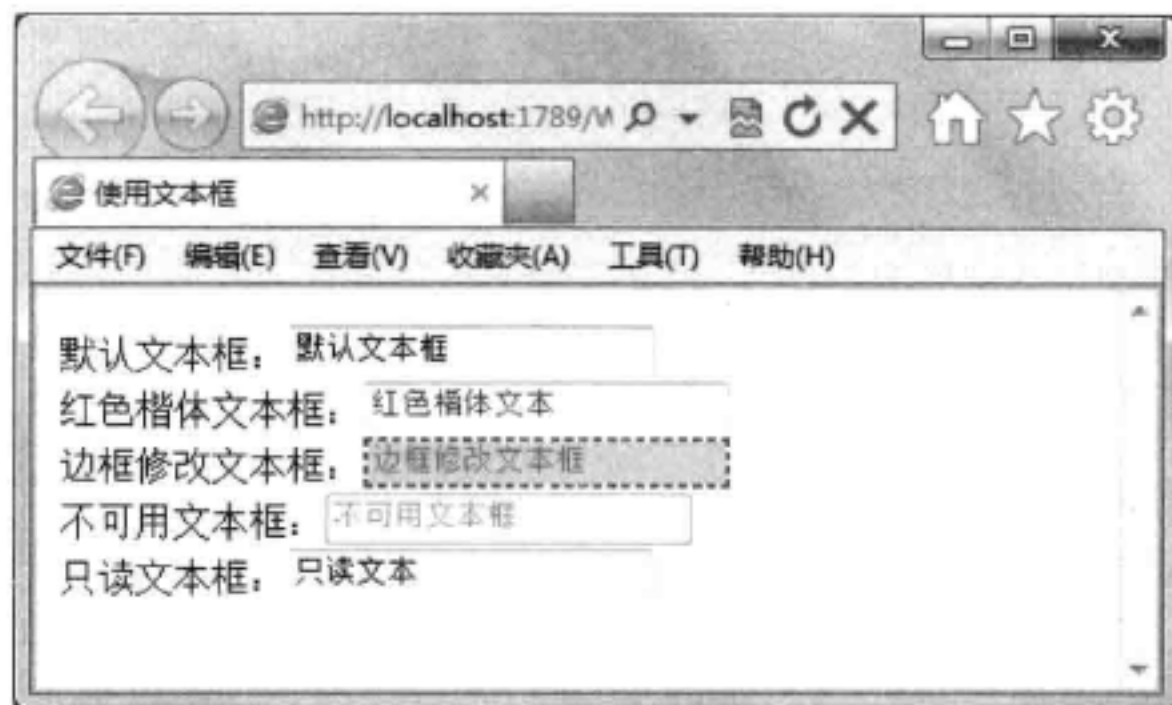


图 25.4 文本框控件效果

25.2.2 用 Button 控件显示按钮

在网页中要实现按钮功能，可以使用 Button 和 LinkButton 控件。Button 控件在网页上由脚本<asp:Button>声明，表现为一个按钮，可以为它配置不同的边框属性（宽度、类型、颜色）和文本属性（前景色、背景色、字体等）得到不同的样式和风格。LinkButton 和 Button 控件一样，也可以用做一个普通的按钮，但是它以一个超链接的形式显示，由脚本<asp:LinkButton>声明。

作为按钮控件，Button 和 LinkButton 都包含 Click 事件，并将事件传回到服务器进行处理，通过 OnClick 属性来声明事件处理函数。下面代码就声明一个 Button 控件，名称 (ID) 为 Button1，显示文本 (Text) 为“按钮一”，Click 事件处理函数 (OnClick) 为 Button1_Click。这样当用户在网页上单击 Button1 按钮时，服务器会调用 Button1_Click 函数，对用户的请求进行处理。

```
<asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
    Text="按钮一" Width="113px" />
```

LinkButton 控件的声明和事件绑定与 Button 控件类似，设置控件属性和绑定事件都可以通过“属性管理器”的属性和事件两个面板可视化完成，Visual Studio 会自动生成代码。下面代码中，前半部分是网页的 ASP.NET 脚本部分，它声明了两个 Button 控件和一个 LinkButton 控件。后半部分是用 C# 语言实现的后台代码，它包含控件的事件处理函数 Button1_Click()、Button2_Click() 和 LinkButton1_Click()，它们分别在一个 Label 控件上显示提示文本。

```
<%@ Page Language="C#" AutoEventWireup="true" %>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>使用按钮控件</title>
</head>
```

```

<body>
  <form id="form1" runat="server">
    <div>
      <asp:Button ID="Button1" runat="server" Text="按钮一" Width="113px"
        onclick="Button1_Click" />
      <asp:Label ID="Label1" runat="server"></asp:Label>
      <br />
      <asp:Button ID="Button2" runat="server" BackColor="#FFFF80"
        BorderColor="Red" BorderStyle="Double" BorderWidth="3px"
        Text="按钮二" Width="112px"
        onclick="Button2_Click" />
      <asp:Label ID="Label2" runat="server"></asp:Label>
      <br />
      <asp:LinkButton ID="LinkButton1" runat="server"
        Width="111px" onclick="LinkButton1_Click">
        超链接按钮一</asp:LinkButton>
      <asp:Label ID="Label3" runat="server"></asp:Label>
    </div>
  </form>
</body>
</html>

```

按钮的代码如下:

```

<script runat="server">
  protected void Button1_Click(object sender, EventArgs e)
  {
    this.Label1.Text = "按钮一 单击! ";
  }

  protected void Button2_Click(object sender, EventArgs e)
  {
    this.Label2.Text = "按钮二 单击! ";
  }

  protected void LinkButton1_Click(object sender, EventArgs e)
  {
    this.Label3.Text = "超链接按钮一 单击! ";
  }
</script>

```

生成并浏览该网页,得到如图 25.5 所示的网页效果,其中“按钮一”是普通样式,“按钮二”则修改了边框宽度、类型和背景色等。“超链接按钮一”则是普通样式,单击前后显示会自动变化。



图 25.5 按钮控件效果图

25.2.3 用 HyperLink 控件显示超链接

在网站开发中,网页之间的跳转或网页内部跳转都是常用功能。在 ASP.NET 中可以通过 HyperLink 控件来实现超链接功能。除了可以设置 HyperLink 的字体、背景色、前景色等外观之外,HyperLink 包含以下两个主要属性。

- **NavigateUrl**: 该属性表示要链接到的目标网页,可以是同一个网站内的页面,如前面的 Default.aspx,也可以是外部网站的页面,如 <http://www.126.com>。
- **Target**: 表示目标网页的打开方式,包括以下 5 个可选值。
 - **_blank**: 将内容呈现在一个没有框架的新窗口中。
 - **_parent**: 将内容呈现在上一个框架的父级中。
 - **_search**: 在搜索窗体中呈现内容。
 - **_self**: 将内容呈现在含焦点的框架中。
 - **_top**: 将内容呈现在没有框架的全窗体中。

合理使用 Target 属性,可以让网站的使用变得简单方便。下面代码中定义了 3 个 HyperLink 控件,分别浏览到前面几节实现的 3 个网页,从它们的 NavigateUrl 属性值可以看出,目标 URL 可以是相对的“~/Default.aspx”,当然也可以是绝对的。生成并运行该示例,可以看到当浏览到“使用 TextBox 控件”网页时,它会在新窗口中打开,因为它的 Target 属性是 _blank。

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>使用超链接</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:HyperLink ID="HyperLink1" runat="server" NavigateUrl=
        "~/Default.aspx" Width="93px">欢迎主页</asp:HyperLink>
      <br />
      <asp:HyperLink ID="HyperLink2" runat="server" NavigateUrl=
        "~/UseButton.aspx" Target="_top"
        Width="123px">使用 Button 控件</asp:HyperLink>
      <br />
      <asp:HyperLink ID="HyperLink3" runat="server" NavigateUrl=
        "~/UseTextBox.aspx" Target="_blank"
        Width="123px">使用 TextBox 控件</asp:HyperLink>
    </div>
  </form>
</body>
</html>
```

25.2.4 用 DropDownList、ListBox 等显示下拉列表、列表

在网页控件中用列表显示数据的控件通常有 DropDownList 和 ListBox 两个控件。DropDownList 控件和 Form 窗体中的 ComboBox 控件类似,以下拉列表的方式显示可选择数据,并提供选中索引变化(SelectIndexChanged)事件。通过 DropDownList 控件的 Items 属性可以添加和删除列表中的元素,并且还可以将它的 Items 属性静态绑定到数据库等数

据源。

网页上的 ListBox 控件是对多个可选项进行选择的最好方式，可以是单选（Single），也可以是多选（Multiple）。通过 ListBox 的 Items 属性也可以添加和移除列表中的可选项。当然，同样可以设置 ListBox 和 DropDownList 控件的字体、背景色、前景色和边框等外观样式。

下面代码中，定义了一个 DropDownList 控件、一个 ListBox 控件和两个 Button 控件，实现将 DropDownList 控件中的选中项添加到 ListBox，将 ListBox 控件中第一个选中项移除的功能。

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>使用列表框</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:DropDownList ID="DropDownList1" runat="server" Width="83px"
                BackColor="#FFC080">
                <asp:ListItem>星期一</asp:ListItem>
                <asp:ListItem>星期二</asp:ListItem>
                <asp:ListItem>星期三</asp:ListItem>
                <asp:ListItem>星期四</asp:ListItem>
                <asp:ListItem>星期五</asp:ListItem>
                <asp:ListItem>星期六</asp:ListItem>
                <asp:ListItem>星期日</asp:ListItem>
            </asp:DropDownList>
            <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
                Text="添加" />
            <br />
            <asp:ListBox ID="ListBox1" runat="server" Height="127px"
                SelectionMode="Multiple"
                Width="84px" ForeColor="Red"></asp:ListBox>
            <asp:Button ID="Button2" runat="server" OnClick="Button2_Click"
                Text="移除" /></div>
        </form>
    </body>
</html>
<script runat="server">
{
    protected void Button1_Click(object sender, EventArgs e)
    {
        //如果下拉列表框中没有选择，则退出
        if (this.DropDownList1.SelectedItem == null)
        {
            return;
        }
        //将选中项添加到列表
        this.ListBox1.Items.Add(this.DropDownList1.SelectedItem);
    }
    protected void Button2_Click(object sender, EventArgs e)
    {
        //移除第一个选中项
        this.ListBox1.Items.Remove(this.ListBox1.SelectedItem);
    }
}
</script>
```


运行代码，得到如图 25.6 所示的效果图，其中 DropDownList 控件的背景色被改成黄色，ListBox 控件的前景色被改成红色。

25.2.5 用 Menu 控件显示导航菜单

菜单导航是网站开发中的一个重要功能，它可以让网站结构更清晰，浏览者可以根据菜单查看感兴趣的网页等，对界面友好性有很大帮助。ASP.NET 中也提供了 Menu 控件来实现网站上的菜单。

可以为 Menu 控件设置不同的外观样式，可以修改它的背景色、前景色、字体等外观属性。一个菜单通常包含多个分层的菜单项，这些菜单项可以从一个数据源动态绑定，也可以通过菜单编辑器手动设置，如图 25.7 所示，每个菜单项可以设置它的可用性（Enable）、显示图片（ImageUrl）、目标网页（NavigateUrl）、弹出图片（PopOutImageUrl）、显示文本（Text）、目标网页打开方式（Target）等属性，合理地设置这些属性，尤其是显示图片等外观属性，可以让网页的菜单非常漂亮和友好。

另外，在 Web 页面中，菜单本身具有横排和竖排两种显示方向（Form 窗体只能横排）。可以通过 Menu 控件的 Orientation 属性来设置它的显示方向，它包括 Vertical 和 Horizontal 两个可选值。

在下面代码中，定义了一个 Menu 控件 Menu1，用来导航前面几节创建的几个示例网页，从代码中 Menu1 的 Items 属性的定义可以看出它包含 5 个一级菜单，其中子菜单“欢迎页面”又包含一个二级菜单“欢迎子菜单”。另外，还添加一个 Button 控件 Button1，在它的 Click 事件处理函数 Button1_Click() 中实现菜单 Menu1 显示方向的自动切换功能。



图 25.6 列表控件效果图

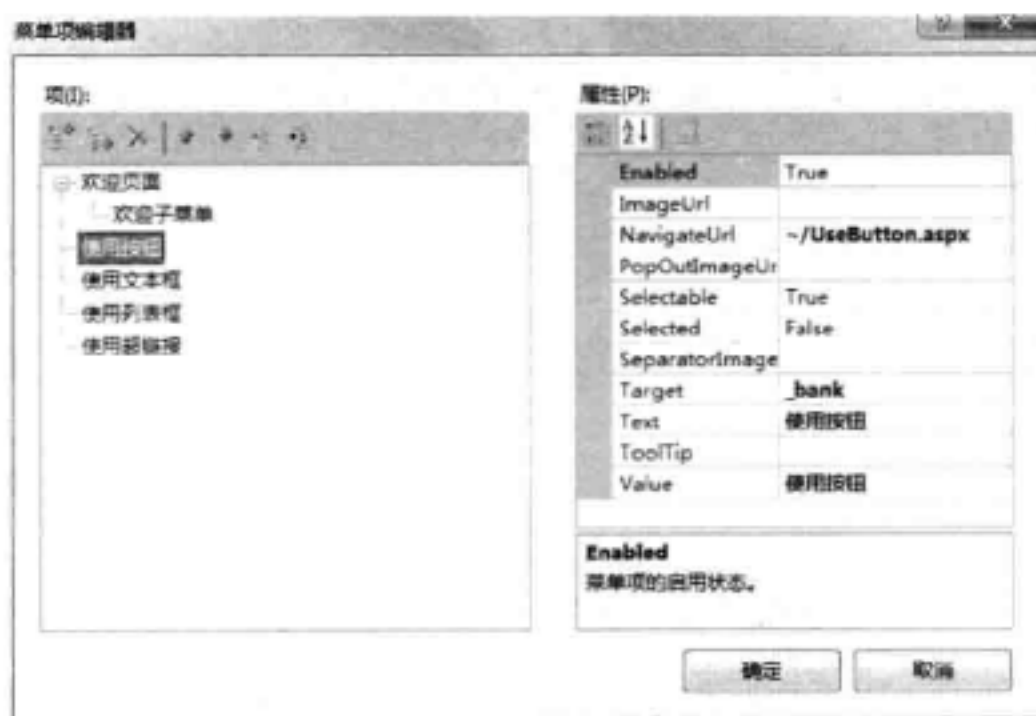


图 25.7 Web 菜单编辑器

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>使用菜单</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Menu ID="Menu1" runat="server" BorderStyle="Double"
                BorderWidth="2px">
```

```

<Items>
  <asp:MenuItem NavigateUrl="~/Default.aspx" Target="_bank"
    Text="欢迎页面" Value="欢迎页面">
    <asp:MenuItem Text="欢迎子菜单" Value="欢迎子菜单">
    </asp:MenuItem>
  </asp:MenuItem>
  <asp:MenuItem NavigateUrl="~/UseButton.aspx" Target="_bank"
    Text="使用按钮" Value="使用按钮">
  </asp:MenuItem>
  <asp:MenuItem NavigateUrl="~/UseTextBox.aspx" Target=
    "_bank" Text="使用文本框" Value="使用文本框">
  </asp:MenuItem>
  <asp:MenuItem NavigateUrl="~/UseList.aspx" Target="_bank"
    Text="使用列表框" Value="使用列表框">
  </asp:MenuItem>
  <asp:MenuItem Target="_bank" Text="使用超链接" Value="使用超链
    接"></asp:MenuItem>
</Items>
</asp:Menu>
</div>
<asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
  Text="改变方向" />
</form>
</body>
</html>

<script runat="server">
  protected void Button1_Click(object sender, EventArgs e)
  {
    //改变菜单显示方向
    if (this.Menu1.Orientation == Orientation.Horizontal)
    {
      this.Menu1.Orientation = Orientation.Vertical;
    }
    else
    {
      this.Menu1.Orientation = Orientation.Horizontal;
    }
  }
</script>

```

运行程序，可以得到如图 25.8 所示的网页。这是 Menu1 竖排显示的效果，同时显示“欢迎页面”菜单的子菜单“欢迎子菜单”。



图 25.8 菜单控件运行效果

25.3 留言板网站实例

前面介绍了常用的 Web 控件，本节以简单的留言本网站作为实例，介绍一个 Web 网站的开发过程，以进一步熟悉用 ASP.NET 开发网站的基本步骤和相关细节。

25.3.1 数据库和页面设计

准确地说，一个留言本并不能算得上是一个网站，因为它通常只是大型网站必备的一个基本模块而已，本节把它作为示例只是对 ASP.NET 网站开发的流程进行讲解。留言本网站通常包括以下两个基本功能。

(1) 访问者查看已有留言：通过该功能，访问者可以浏览已有的留言信息，可以从中找到一些有用的信息等，这就要求网站具有保存历史留言信息的能力。

(2) 访问者添加新的留言：访问者输入新的留言信息，通过网站保存到数据库中，输入留言信息同时需要输入留言者、留言主题、留言内容。留言时间也应该要记录，但是不应该让访问者输入，而是程序自动获取当前系统时间并记录。

基于上面两个需求，留言本网站需要设计一个简单的留言数据库——留言信息，由于很简单，只需要使用 Access 数据库即可。留言信息数据库的数据表信息如表 25.1 所示，具有一个数据库表——留言信息，表“留言信息”包含 4 个字段，即留言者、留言时间、留言标题、留言内容。

表 25.1 留言信息数据库表

表 名	字 段 名	字段类型	字 段 说 明
留言信息	留言者	文本	长度为50的文本，表示留言者名称，与留言时间一起作为主键
	留言时间	文本	长度为50的文本，表示留言时间
	留言标题	文本	长度为100的文本，表示留言信息的标题，简单描述留言的目的等
	留言内容	文本	长度为255的文本，表示留言的具体内容，可以多达255个字符

此外，留言本网站还需要设计以下 3 个不同功能的 Web 页面。

(1) 欢迎页面：该页面主要是给出欢迎信息，为了让留言本看起来更像一个网站。

(2) 查看留言页面：该页面让访问者可以查看所有已经存在的留言信息。

(3) 添加留言页面：该页面让访问者可以添加自己的留言信息到数据库，并可以通过“查看留言”页面查看新增的留言信息。

在明确了这些基本问题之后，需要动手创建一个 Web 网站应用程序，具体步骤见 25.1.3 节的描述，大致包括以下 3 个步骤。

(1) 创建一个 Web 网站应用程序，命名为 LiuYan。

(2) 创建一个具有表 25.1 所示的 Access 数据库文件，命名为 LiuYan.mdb，并将它添加到网站 LiuYan 的 App_Data 目录下（App_Data 目录，主要是用来存放网站所需要用到的数据）。

(3) 依次添加“欢迎页面”、“查看留言页面”和“添加留言页面”到应用程序，分

别命名为 welcome.aspx、viewliuyan.aspx 和 addliuyan.aspx。最后得到 Web 应用程序 LiuYan 的文件结构如图 25.9 所示。

25.3.2 设计欢迎页面

一个独立的网站通常都包含一个“主页/欢迎页面”，该页面通常包括本站的使用帮助信息，重要信息汇总等，对访问者起到一个指导和了解本站概况的作用。为了使留言本 Web 应用程序看起来更像一个网站，特地为它添加欢迎页面——Welcome.aspx。

在留言本 Web 应用程序中，Welcom.aspx 主要包含两个部分：导航菜单和说明，它的 ASP.NET 脚本程序如下所示，它的标题为“欢迎使用留言本~~~”。包括名为 Menu1 的菜单，将访问者导航到“添加留言”、“查看留言”和“主页”

3 个页面。Menu1 后面是以简单的文本形式给出的网站的功能描述等信息。



图 25.9 LiuYan 网站的文件结构

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="welcome.aspx.cs"
    Inherits="Welcome" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>欢迎使用留言本~~~</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Menu ID="Menu1" runat="server" ForeColor="#8080FF"
                Orientation="Horizontal" Font-Size="Large" Height="37px" Width=
                "287px">
                <Items>
                    <asp:MenuItem NavigateUrl="~/welcome.aspx"
                        Text="主页" Value="主页"></asp:MenuItem>
                    <asp:MenuItem NavigateUrl="~/addliuyan.aspx"
                        Text="添加留言" Value="添加留言"></asp:MenuItem>
                    <asp:MenuItem NavigateUrl="~/ViewLiuyan.aspx"
                        Text="查看留言" Value="管理留言"></asp:MenuItem>
                </Items>
            </asp:Menu>
            <div>
                <p>
                    <strong><span style="font-size: 14pt; font-family: 华文中宋;
                        background-color: #ff9966">
                        欢迎使用留言本，通过菜单进入对应的功能 </span></strong>
                </p>
                <p>
                    <span style="color: #cc66ff; font-family: 宋体">
                        这是本书中关于 web 网页开发和设计的例子程序，它接收浏览器输入留言信息，
                        并保存到数据库
                    <br />
                </p>
            </div>
        </div>
    </form>
</body>
</html>
```



```

        同时管理者可以对留言信息进行删除等操作 </span>
    </p>
</div>
</div>
</form>
</body>
</html>

```

25.3.3 设计添加留言页面

“添加留言页面”是留言本 Web 应用程序的一个主要页面，它使访问者可以将自己想留下的信息保存到数据库，这就要求该页面具有以下 3 个主要功能。

(1) 页面导航：将访问者导航到“欢迎页面”和“查看留言页面”。

(2) 为访问者提供留言信息输入界面：包括留言者、留言主题、留言内容的输入，留言时间则由网站自动生成，这样可减少访问者输入不必要的信息。

(3) 提供“保存”按钮，让访问者可以保存留言信息到数据库，在保存前要对输入信息合法性进行检查，并给出提示信息。

下面代码是“添加留言页面”addliuyan.aspx 的 ASP.NET 脚本程序，它的标题为“添加留言”。包含导航菜单 Menu1，将访问者导航到“添加留言”、“查看留言”和“主页”3 个页面。接着是一个面板 Panel1，该面板包含 3 个 TextBox 控件，分别是 tbName、tbTitle 和 tbContent，分别用来输入“留言者”、“留言主题”和“留言内容”。最后还提供一个“留言”按钮——btnAdd，该按钮的 Click 事件响应函数为 btnAdd_Click()，负责将留言信息添加到数据库中。

页面文件名全为小写字母，因为在下面的代码中编写了一个 AddLiuYan 的函数，如果页面文件名和函数名相同，则会出错。

```

<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="AddLiuYan.aspx.cs" Inherits="AddLiuYan" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>添加留言</title>
</head>
<body style="font-size: 12pt">
    <form id="form1" runat="server">
        <div>
            <asp:Menu ID="Menu1" runat="server" Font-Size="Large"
                ForeColor="#8080FF" Height="37px"
                Orientation="Horizontal" Width="287px">
                <Items>
                    <asp:MenuItem NavigateUrl="~/welcome.aspx"
                        Text="主页" Value="主页"></asp:MenuItem>
                    <asp:MenuItem NavigateUrl="~/addliuyan.aspx"
                        Text="添加留言" Value="添加留言"></asp:MenuItem>
                    <asp:MenuItem NavigateUrl="~/ViewLiuyan.aspx"
                        Text="查看留言" Value="管理留言"></asp:MenuItem>
                </Items>
            </asp:Menu>

```



```

<span style="color: #009966">
<span style="background-color: #ffccff">
    通过该页面添加新留言到数据库, 单击"添加"按钮添加!!!!</span>
<br />
</span>
</div>
<asp:Panel ID="Panel1" runat="server" Height="333px" Width="557px"
BorderColor="Red" BorderStyle="Dashed" BorderWidth="1px">
    <asp:Label ID="Label1" runat="server" Text="留言者: ">
</asp:Label>
    <asp:TextBox ID="tbName" runat="server" Width="189px">
</asp:TextBox>
    <br />
    <asp:Label ID="Label2" runat="server" Text="标题: " Width="63px">
</asp:Label>
    <asp:TextBox ID="tbTitle" runat="server" Width="476px">
</asp:TextBox>
    <br />
    <hr />
    <asp:Label ID="Label3" runat="server" Text="内容: " Width="63px">
</asp:Label>
    <br />
    <asp:TextBox ID="tbContent" runat="server" Height="192px"
Width="539px"></asp:TextBox>
    <br />
    <asp:Label ID="lbHint" runat="server" ForeColor="Red">
</asp:Label>
    <br />
    <asp:Button ID="btnAdd" runat="server" BackColor="#FFC080"
        BorderColor="#C0FFC0" BorderStyle="Ridge"
        Font-Bold="True" Font-Names="宋体"
        Font-Overline="False" Font-Size="Larger"
        OnClick="btnAdd_Click" Text="留言" Width="112px" />
</asp:Panel>
<br />
</form>
</body>
</html>

```

由于需要在后台进行数据有效性判断和数据库操作, 这些复杂功能不宜在 ASP.NET 的脚本程序中实现, 所以需要在 C# 代码文件 AddLiuYan.aspx.cs 中给出和数据相关的处理代码, 主要是 btnAdd_Click() 方法的实现, 下面代码是“添加留言页面”的主要实现代码。其中, Page_Load() 方法是每次页面加载时 Load 事件的处理函数, 这里通常需要通过 Page 类的 IsPostBack 属性判断页面是否为浏览器重新发回到服务器的页面。如果是, 则需要保留原来页面上的数据, 如本例中 Page_Load() 方法。

btnAdd_Click() 方法则首先清除要提示的内容, 然后从输入框 tbName、tbTitle 和 tbContent 中获取用户输入信息, 并进行合法性判断。如果不合法, 则提示并不会保存到数据, 如果合法, 则通过 AddLiuYan() 方法将信息保存到数据库中。在 AddLiuYan() 方法中, 首先用 OleDbConnection 类创建一个数据库连接对象, 然后创建一个“INSERT INTO”SQL 命令, 最后通过 OleDbCommand 类的 ExecuteNonQuery() 方法将新数据插入到数据库 LiuYan.mdb 中。

```

using System;
using System.Data;
using System.Configuration;

```



```
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Data.OleDb;

public partial class addliuyan : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //不是重新发送回来, 则更新网页界面
        if (!this.IsPostBack)
        {
            this.tbName.Text = "";
            this.tbTitle.Text = "";
            this.tbContent.Text = "";
        }
    }

    //添加留言
    protected void btnAdd_Click(object sender, EventArgs e)
    {
        //更新提示信息
        this.lbHint.Text = "";

        bool ok = true;
        string hint="";
        //判断留言者的合法性
        string name = this.tbName.Text.Trim( );
        if (string.IsNullOrEmpty(name))
        {
            ok = false;
            hint = "留言者不能为空<br/>";
        }
        //判断留言主题合法性
        string title = this.tbTitle.Text.Trim( );
        if (string.IsNullOrEmpty(title))
        {
            ok = false;
            hint += "留言主题不能为空<br/>";
        }
        //判断留言内容合法性
        string content = this.tbContent.Text.Trim( );
        if (string.IsNullOrEmpty(content))
        {
            ok = false;
            hint += "留言内容不能为空<br/>";
        }
        //如果有数据不合法, 提示并退出
        if (!ok)
        {
            this.lbHint.Text = hint;
            this.lbHint.ForeColor = System.Drawing.Color.Red;
            return;
        }
        //保存数据到数据库
    }
}
```




```

        if (this.AddLiuYan(name, title, content))
        {
            //成功提示
            this.lbHint.Text = "添加成功<br/>";
            this.lbHint.ForeColor = System.Drawing.Color.Green;
            this.tbName.Text = "";
            this.tbTitle.Text = "";
            this.tbContent.Text = "";
        }
        else
        {
            //失败提示
            this.lbHint.Text = "添加失败<br/>";
            this.lbHint.ForeColor = System.Drawing.Color.Red;
        }
    }

    private bool AddLiuYan(string name, string tiltle, string content)
    {
        //创建数据库连接, 连接到 LiuYan.mdb Access 数据库
        OleDbConnection dbcon = new OleDbConnection(
            "Provider=Microsoft.Jet.OLEDB.4.0;
            Data Source="+ AppDomain.CurrentDomain.BaseDirectory
            + "\\App_Data\\LiuYan.mdb;Persist Security Info=True");
        //生成 SQL 命令字符串
        string cmdStr = "INSERT INTO [留言信息] VALUES('";
        cmdStr += name + "', '";
        cmdStr += DateTime.Now.ToString() + "', '";
        cmdStr += tiltle + "', '";
        cmdStr += content + "')";
        //创建并执行 SQL 命令
        OleDbCommand cmd = new OleDbCommand(cmdStr, dbcon);
        dbcon.Open();
        cmd.ExecuteNonQuery();
        //关闭数据连接
        dbcon.Close();
        return true;
    }
}

```

 **注意：**浏览器客户端向服务器每发送一次请求，表示页面的 Page 类（如上面的 addliuyan 类）都是重新创建的，所以页面类的所有私有变量都不再有效。

生成并运行添加留言页面，得到如图 25.10 所示的效果。读者可以在原有代码基础上，对该页面进行美化，使得它看起来更加自然和友好。

25.3.4 设计查看留言页面

“查看留言页面”是留言本 Web 应用程序的另外一个主要页面，它让访问者可以查看当前已经存在的留言信息。该页面的主要功能是从数据库获取留言信息，并逐个显示到网页上，所以“查看留言页面”主要包括以下几部分。

- ☐ 页面导航：将访问者导航到“欢迎页面”和“查看留言页面”。
- ☐ “查看”按钮：访问者通过该按钮刷新当前已经存在的留言信息。
- ☐ 留言信息：逐个显示从数据库读取到的留言信息。



图 25.10 添加留言页面效果图

下面的代码是“查看留言页面”的 ASP.NET 脚本程序，它包括一个名为 Menu1 的导航菜单。包含一个名为 btnView 的“查看”按钮，该按钮的 Click 事件响应函数为 btnView_Click()。最后还包括一个名为 lbLiuYan 的 Label 控件（和 Form 窗体中的 Label 控件一样，用来显示静态文本），用来显示读取到的留言信息。

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="ViewLiuYan.aspx.cs" Inherits="ViewLiuYan" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>查看留言</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Menu ID="Menu1" runat="server" Font-Size="Large"
            ForeColor="#8080FF" Height="37px"
            Orientation="Horizontal" Width="287px">
            <Items>
                <asp:MenuItem NavigateUrl="~/welcome.aspx" Text="主页"
                    Value="主页"></asp:MenuItem>
                <asp:MenuItem NavigateUrl="~/addliuyan.aspx"
                    Text="添加留言" Value="添加留言"></asp:MenuItem>
                <asp:MenuItem NavigateUrl="~/ViewLiuyan.aspx"
                    Text="查看留言" Value="管理留言"></asp:MenuItem>
            </Items>
        </asp:Menu>
        <br />
        <asp:Button ID="btnView" runat="server" BackColor="#FFC080"
            BorderColor="#C0C0FF" BorderStyle="Solid" BorderWidth="1px"
            Font-Size="X-Large" ForeColor="Green" Height="30px"
            OnClick="btnView_Click" Text="查看" Width="108px" />
    </div>
    </form>
</body>
</html>
```

```

        <br />
        <asp:Label ID="lbLiuYan" runat="server"></asp:Label>
    </div>
</form>
</body>
</html>

```

和“添加留言页面”一样，ViewLiuYan 页面也需要访问数据库，同样需要通过后台的 C# 代码文件 viewliuyan.aspx.cs 来实现数据处理，主要包括 btnView_Click() 方法。下面的代码是“查看留言页面”的主要实现代码，其中 btnView_Click() 方法通过 LoadLiuYan() 方法获取数据库信息并显示到 lbLiuYan 上。LoadLiuYan() 首先用 OleDbConnection 类创建一个数据库连接对象，然后创建一个“SELECT”SQL 命令，接下来通过 OleDbCommand 类的 ExecuteReader() 方法从数据库 LiuYan.mdb 中获取一个 DataReader 对象 dr，最后通过 dr 依次读取所有的留言信息，并通过 ShowLiuYan() 方法显示到 lbLiuYan 控件上。在 ShowLiuYan() 方法中，按照固定格式将留言信息组织成一段 HTML 文本，然后显示到 lbLiuYan 标签上。

```

public partial class ViewLiuYan : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }

    //查看按钮事件响应函数
    protected void btnView_Click(object sender, EventArgs e)
    {
        //加载所有的留言信息
        this.LoadLiuYan();
    }

    private void LoadLiuYan()
    {
        //打开数据库连接，连接到 LiuYan.MDB
        OleDbConnection dbcon = new OleDbConnection(
            "Provider=Microsoft.Jet.OLEDB.4.0;
            Data Source=" + AppDomain.CurrentDomain.BaseDirectory
            + "\\App_Data\\LiuYan.mdb;Persist Security Info=True");
        //查询 SQL 命令
        string cmdStr = "SELECT [留言者], [留言时间], [留言标题], [留言内容] FROM [留言信息]";
        //创建 SQL 命令
        OleDbCommand cmd = new OleDbCommand(cmdStr, dbcon);
        dbcon.Open();
        //以 DataReader 方式执行 SQL 语句
        OleDbDataReader dr = cmd.ExecuteReader();
        //清空留言显示信息
        this.lbLiuYan.Text = "";
        //循环读取所有的留言信息
        while (dr.Read())
        {
            //读取留言信息
            string name = (string) dr["留言者"];
            string time = (string) dr["留言时间"];
            string title = (string) dr["留言标题"];
            string content = (string) dr["留言内容"];
            //显示留言信息到界面

```



```

        this.ShowLiuYan(name, time, title, content);
    }
    //关闭数据库连接
    dbcon.Close();
}

private void ShowLiuYan(string name, string time, string title,
    string content)
{
    //生成留言信息显示的 HTML 文本段
    string lyStr = string.Format("<P><b>留言者:</b>{0},<b>留言时间:</b>{1}<br><b>标题:</b>{2}<br><b>内容:</b><br>{3}<br></P>",
        name,
        time,
        title,
        content);

    //显示到界面
    this.lbLiuYan.Text += lyStr;
}
}

```

生成并运行“查看留言页面”，得到如图 25.11 所示的效果。读者可以在原有代码基础上，对该页面进行美化，使它看起来更加自然和友好。



图 25.11 查看留言页面效果图

25.3.5 发布留言板网站

到此为止，已经创建好了一个简单的留言板网站，并且在调试环境下可以正常运行了，接下来要做的就是如何将网站展示给访问者，这就需要发布网站了。

在 Visual Studio 2010 要发布网站只需要以下 6 个简单的步骤就可以完成。

(1) 确保 Web 应用程序没有错误，即至少可以调试正常运行。确保操作系统上已经安装了 IIS 服务器组件。

(2) 在“解决方案管理器”中选中 Web 项目并右击鼠标（本例中选中 LiuYan），在弹出菜单中选择“发布网站”命令，弹出“发布网站”对话框。

(3) 在该对话框的“目标位置”栏单击“...”按钮，弹出“选择发布位置”对话框，

如图 25.12 所示,选择目标位置为“本地 IIS”,并选择 Default Web Site,然后单击“打开”按钮,回到“发布网站”对话框。



图 25.12 “选择发布位置”对话框

(4) 在“发布网站”对话框中单击“确定”按钮,开始发布,可以在“输出”窗口看到发布过程,大致会得到如下所示的过程提示。

```
----- 已启动生成: 项目: d:\...\LiuYan\, 配置: Debug Any CPU -----
正在预编译网站

正在生成目录 “/LiuYan/”。
预编译完成
----- 发布已启动: 项目: d:\...\LiuYan\, 配置: Debug Any CPU -----
正在连接到站点 http://localhost...
正在删除现有文件...
正在发布目录 /...
正在发布目录 App_Data...
正在发布目录 bin...
===== 生成: 成功或最新 1 个, 失败 0 个, 跳过 0 个 =====
===== 发布: 成功 1 个, 失败 0 个, 跳过 0 个 =====
```

(5) 通过上面 4 个步骤完成网站的发布之后,可以在“计算机管理”的 IIS 服务器目录下看到前面开发的网页,如图 25.13 所示,例如 welcome.aspx、viewliuyan.aspx 等,另外,App_Data 文件夹还包括数据库文件 LiuYan.mdb。Bin 文件夹下是自动生成的 DLL 文件,它们与 aspx 脚本文件一起共同组成了该网站的所有程序,此时已经可以通过 http://127.0.0.1/welcome.aspx 访问欢迎页面,但是网址 http://127.0.0.1 还不能正常访问。



图 25.13 “计算机管理”对话框

(6) 为了让 `http://127.0.0.1` 能正常访问，还需要再设置 IIS 服务器的默认文档属性。在图 25.13 的对话框中单击中下方的“功能视图”选项，切换到功能列表视图，在视图的“IIS”组中找到并双击“默认文档”图标，切换到默认文档设置界面，然后单击右上角的“添加”按钮，将留言本网站的默认网页 `welcome.aspx` 添加到列表中，如图 25.14 所示。此时可以通过 `http://127.0.0.1` 访问留言板网站。



图 25.14 设置默认启动页

25.4 本章总结

本章简单介绍了 ASP.NET 开发 Web 应用程序的基本概念，并介绍了 ASP.NET 与 C# 集成开发的方法，还介绍了 ASP.NET 开发中常用的 Web 服务器控件，以及它们的使用示

例。最后通过一个简单的留言板网站实例介绍了网站从设计到开发，最后发布的过程。

25.5 实战练习

1. 在 Visual Studio 2010 中新建一个 ASP.NET 网站，修改生成的 Default.aspx，在网页中显示“我的第一个 ASP.NET 网站”。

2. 在 Visual Studio 2010 中新建一个 ASP.NET 空网站，向网站中添加一个 Register.aspx 页面，在该页面中添加一个 TextBox 控件、一个 Button 控件，当单击 Button 控件时，在下方显示 TextBox 控件中输入的内容。

3. 在 Visual Studio 2010 中新建一个 ASP.NET 空网站，制作 25.3 节所介绍的留言板，并美化相关的页面。

第 26 章 服务器端控件详解

由于 Web 应用程序是建立在浏览器/服务器 (B/S) 结构的基础上, 因此 Web 程序不但要求能够处理客户端事务, 而且要求能够与服务器端实现交互。为了能够很好地适应这种浏览器端和服务器端的工作模式, 微软公司在 ASP.NET 中创建了很多专门运行于服务器端的 Web 控件, 这种控件又叫做 ASP.NET 控件。可以使用 Web 控件创建服务器端代码, 以响应在客户端上发生的事件。

26.1 认识服务器控件

实际上, 服务器控件可以分为 HTML 控件和 ASP.NET 控件两大类。对 HTML 控件而言, 通常用到的都是客户端的控件。如果要使该控件能够运行于服务器端, 则需要在其属性设置中添加 `runat=server`。

26.1.1 为什么使用服务器控件

服务器控件是相对于客户端控件而言的。客户端控件是指运行于客户端的 HTML 控件; 而服务器控件则包含运行于服务器端的 HTML 控件和 ASP.NET 控件两大类。

前面说到, 如果要使 HTML 控件能够运行于服务器端, 需要将 `runat=server` 添加到它的属性中。此时该 HTML 控件的左上角会出现一个三角符号, 其外形与 ASP.NET 控件是一样的。定义 HTML 控件的类位于 `System.Web.UI.HtmlControls` 命名空间中。

之所以还要开发出 ASP.NET 控件, 是因为 ASP.NET 控件使 HTML 控件更为简洁, 它提供的功能更为丰富; 并且在 ASP.NET 中提供的控件不但包括文本框、按钮、表格等窗体类型的控件, 而且还提供了更多具有高级功能的控件, 如 `GridView`、`Calendar`、`DataList` 和 `RequiredFieldValidator` 等控件。在服务器端, ASP.NET 控件需要经过解析后, 转换为相应的 HTML 控件再传给客户端, 而服务器端 HTML 控件可以直接传给客户端。定义 ASP.NET 控件的类位于 `System.Web.UI.WebControls` 命名空间中。

可以将 HTML 控件和 ASP.NET 视为对象, 它们又有各自的属性、方法和事件。

26.1.2 服务器控件与 HTML 控件的区别

ASP.NET 控件运行于服务器端, 是服务端控件, 响应服务端事件。HTML 控件是客户端控件, 响应客户端事件。

简单地说, 使用客户端 HTML 控件所产生的事件是不会提交给服务端的。如单击一个

按钮改变文字的颜色，只是针对用户机器本身，不会发送数据信息给远程的服务器处理。而使用服务器控件按钮编写的程序，单击后将会以 POST 或 GET 形式发送给服务器进行处理，同时页面提交后客户端将会刷新服务器端发送的新页面。

26.2 数据操作控件

在 ASP.NET 4 中提供了许多非常强大的数据库操作控件。使用这些控件可以轻松地创建出功能强大的数据库操作应用程序。这些控件包括 GridView、DataList、DetailsView、SqlDataSource 和 FormView 等。本节将介绍一些常用的数据操作控件。

26.2.1 SqlDataSource 的作用

SqlDataSource 控件在数据库的操作中，起着桥梁的作用。它连接了数据库和用于显示数据库中内容的控件。通过该控件，可以设置访问数据库的方法、显示数据的方法等属性。它常与 GridView 和 DetailView 等控件一起使用。用于操作数据库的这些控件位于“工具箱”中的“数据”栏中，如图 26.1 所示。

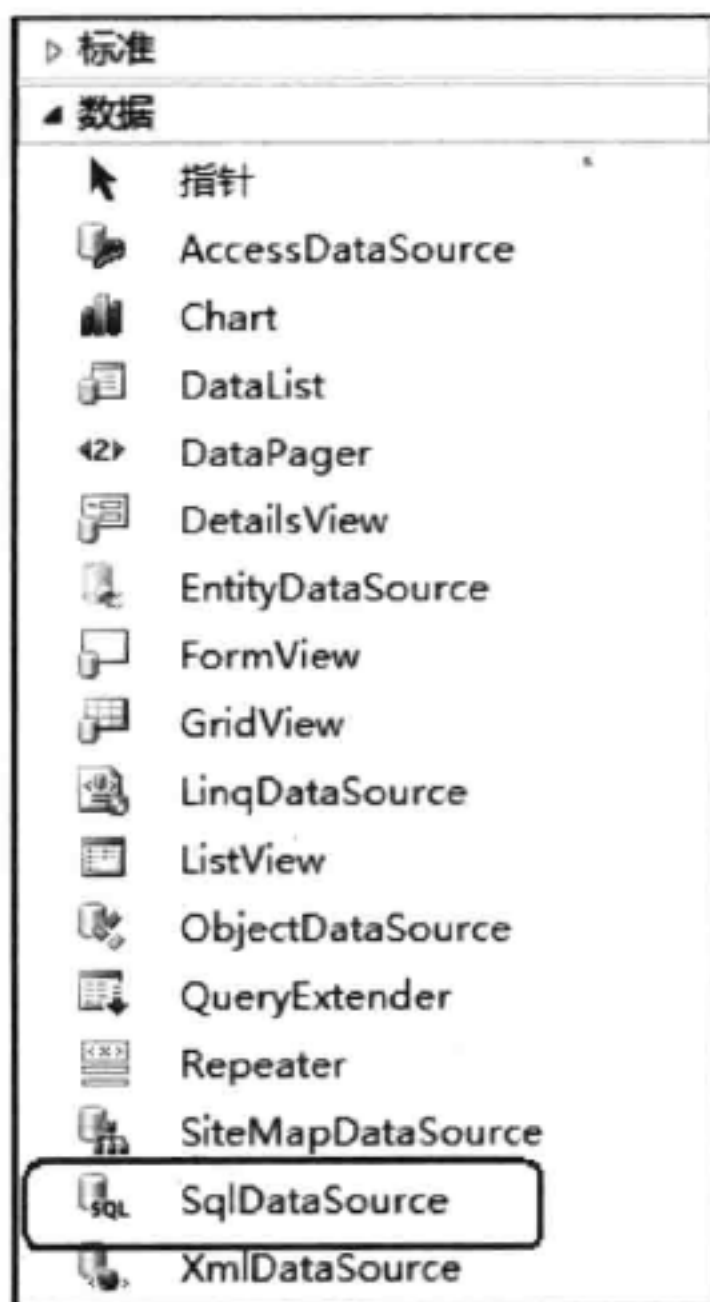


图 26.1 数据控件

SqlDataSource 控件可以与使用 ADO.NET 类，以及 ADO.NET 支持的任何数据库进行数据交互。这些数据库包括 Microsoft SQL Server、Oracle 数据库，以及 System.Data.OleDb 和 System.Data.Odbc 数据源。

使用 SqlDataSource 控件，可以在 Web 页中访问和操作数据，而无需使用 ADO.NET

提供的类。当然，这需要提供用于连接到数据库的连接字符串，同时定义操作数据的 SQL 语句或存储过程。在运行时，SqlDataSource 控件会自动打开数据库连接，执行其中指定的 SQL 语句或存储过程，最后返回执行结果，并关闭连接。

26.2.2 用 SqlDataSource 控件连接到数据库

在 Visual Studio 2010 中，使用很少的代码或不使用代码就可以实现在网页中显示和操作数据及数据库。下面将介绍通过 SqlDataSource 连接到数据库的方法。

(1) 打开 Visual Studio 2010，新建一个 ASP.NET 空 Web 应用程序，并命名为 SqlDataSourceTest。

(2) 向网站中添加一个名为 Default.aspx 的页面，页面切换到“设计”视图。在工具箱中的“数据”栏里找到 SqlDataSource 控件，并将其拖放到页面中，如图 26.2 所示。



图 26.2 SqlDataSource 控件

(3) 在图中的右边出现了“SqlDataSource 任务”，单击“配置数据源”链接，弹出如图 26.3 所示“配置数据源”对话框。下面为其创建一个数据库连接。



图 26.3 “配置数据源”对话框

本例中使用的 AdventureWorks2008 数据库是 SQL Server 2008 的演示数据库，读者可从微软网站下载，然后附加（或还原）到本机 SQL Server 2008 中就可使用了。

(4) 单击“新建连接(C)…”按钮,在“选择数据源”对话框中选择数据源为 Microsoft SQL Server。然后单击“继续”按钮,弹出如图 26.4 所示“添加连接”对话框,在“服务器名”项对应的下拉菜单中选择自己所建立的数据库服务器,本文例程为“\SQLEXPRESS”。这时“连接到一个数据库”一栏中的两个单选按钮处于可选状态,在“选择或输入一个数据库名”对应的下拉列表框中选择一个带有实例的数据库,如“AdventureWorks2008”。单击“测试连接”按钮,系统会弹出一个测试成功提示框,表示可以操作数据库。单击“确定”按钮,返回到如图 26.3 所示界面,只是其中的“应用程序连接数据库应使用哪个数据库连接?”对应的下拉选项被选为“wyhnp\sqlcxpress. AdventureWorks2008.dbo”。

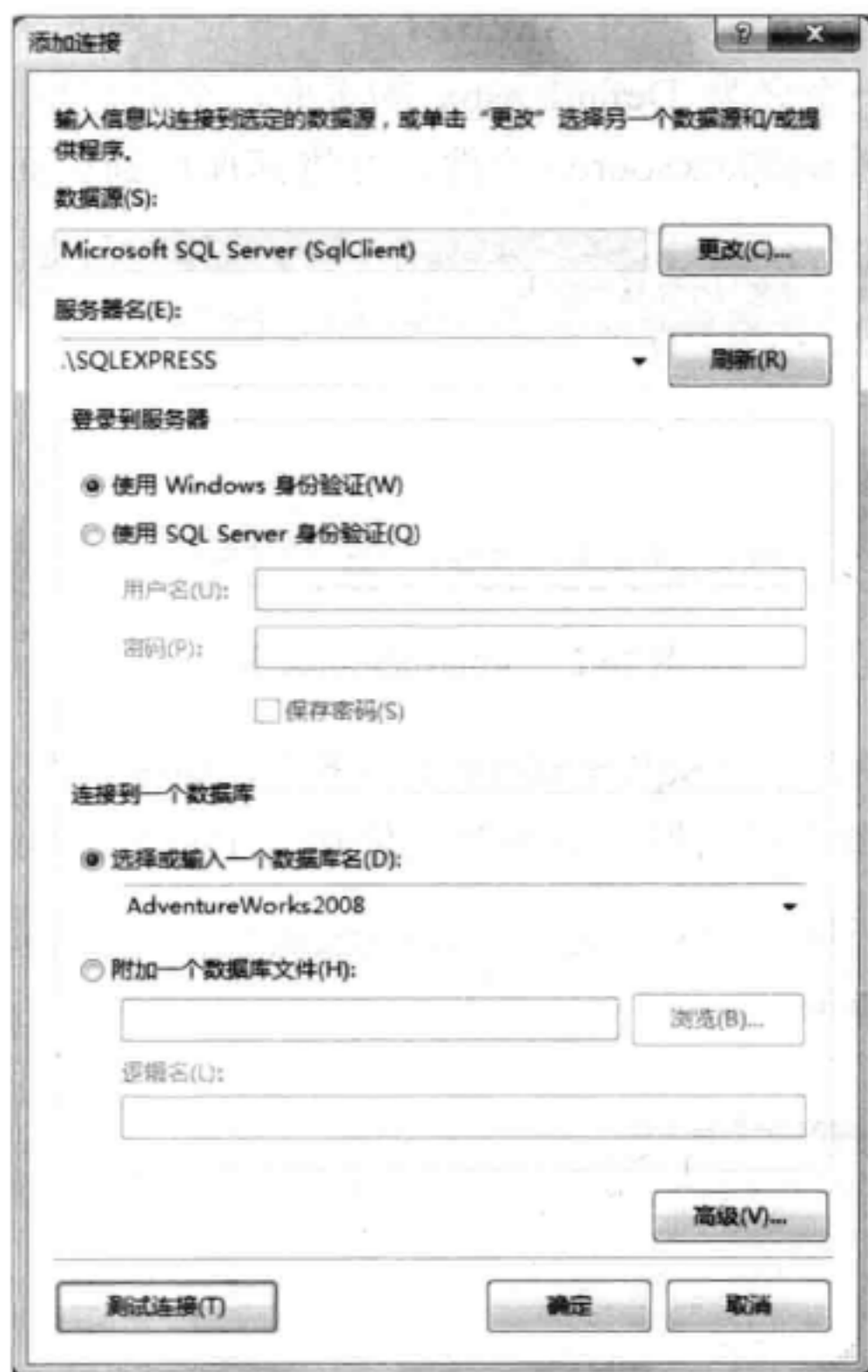


图 26.4 “添加连接”对话框

(5) 展开如图 26.3 所示对话框中的“连接字符串”节点,可以看到其自动建立的数据库连接字符串,如下所示。

```
Data Source=.\SQLEXPRESS;Initial Catalog=AdventureWorks 2008;Integrated Security=True
```

(6) 单击“下一步”按钮,弹出如图 26.5 所示的对话框。该对话框中提示是否将连接字符串保存到应用程序配置文件(Web.Config)中,如保持默认则勾选该项。

(7) 单击“下一步”按钮,弹出“配置数据源”对话框,选定其中的“指定来自表或视图的列”单选项,在“名称”对应的下拉列表框中选择 AdventureWorks2008 数据库自带的表 DatabaseLog,如图 26.6 所示。

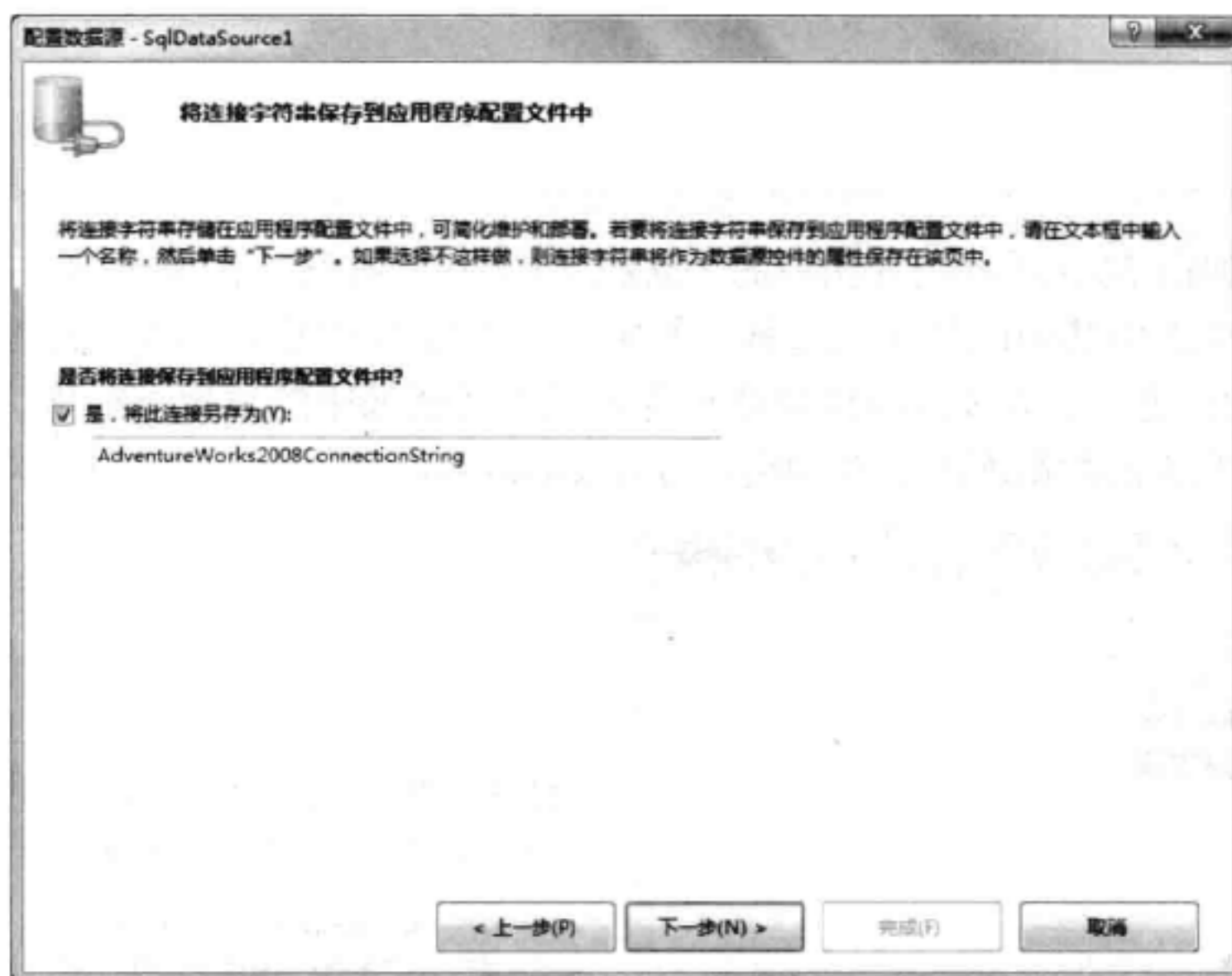


图 26.5 保存到配置文件



图 26.6 配置数据检索方式

(8) 在“列”列表框中可以看到，其中列出了该表中所有的列。选中其中任意几个列，如 DatabaseLog ID、PostTime、DatabaseUser 和 Event，可以看到 SELECT 语句文本框自动生成了相应功能的查询语句。在“列”列表框的右边“只返回唯一行”表示，如果查询结果是多行，则只返回其中的第一行。WHERE (W) 用于配置查询条件。再单击 ORDER BY (R) 按钮，得到配置排序对话框。选择“排序方式”中的 DatabaseLog ID，如图 26.7 所示。

保持默认的“升序”排列，并单击“确定”按钮，可以看到系统自动生成如下所示的

查询语句。

```
SELECT [DatabaseLogID] , [PostTime], [DatabaseUser], [Event] FROM
[DatabaseLog] ORDER BY [DatabaseLogID]
```

(9) 单击如图 26.6 所示的界面中的“高级 (V) …”按钮，在弹出的“高级 SQL 生成选项”对话框中选中其中的两个复选框。其中第一个复选框用于自动生成具有增、删、改功能的附加 SQL 语句。通过这些语句命令便可以对数据源中的数据进行增、删、改，而第二个复选框用于防止数据库的并发冲突，如图 26.8 所示。



图 26.7 设置排序方式



图 26.8 高级 SQL 生成选项

(10) 单击“确定”按钮，返回到“配置数据源”对话框，单击“下一步”按钮，弹出“测试查询”对话框。单击“测试查询”按钮，可以预览该数据源返回的数据，如图 26.9 所示。

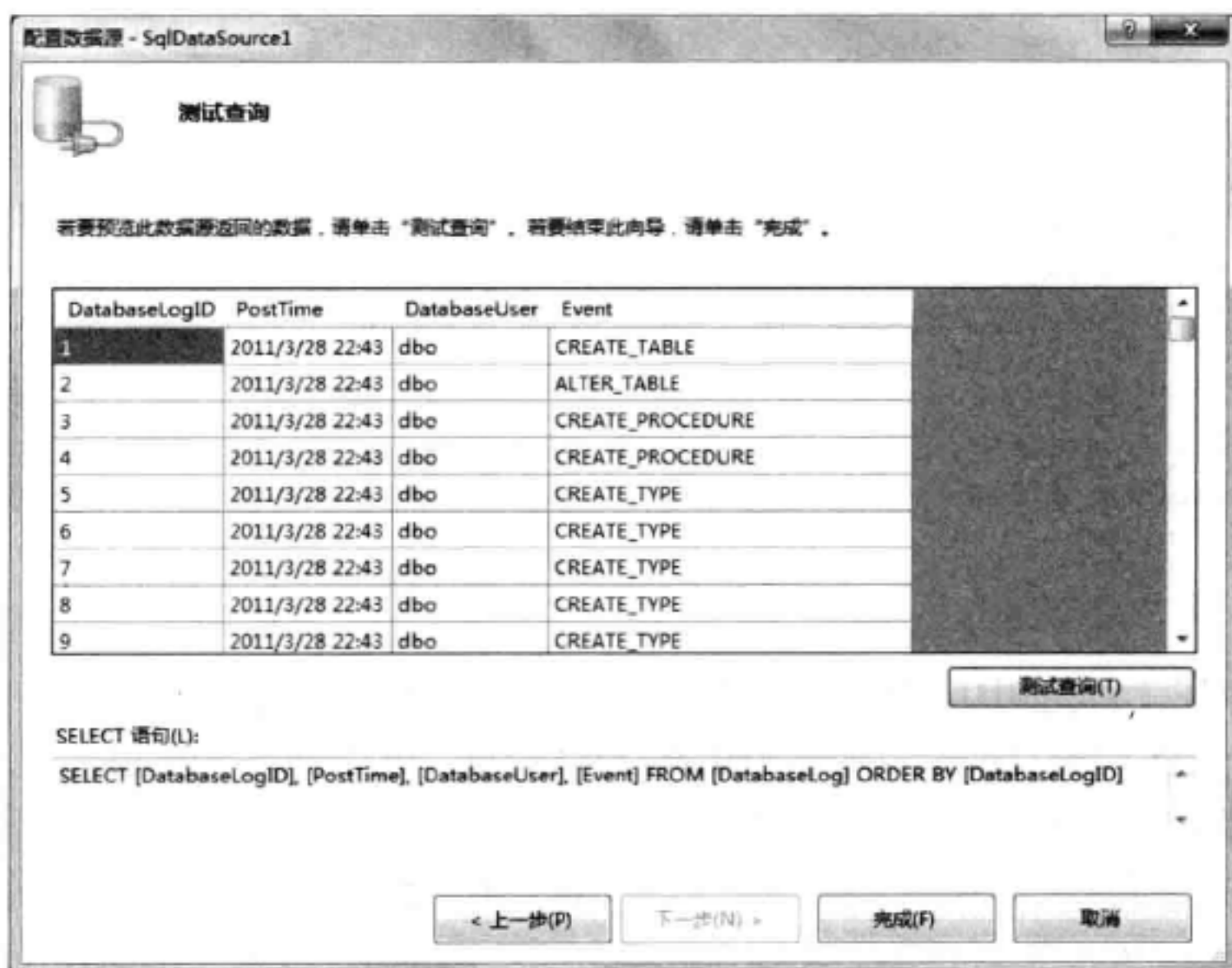


图 26.9 测试查询

(11) 单击“完成”按钮，完成了数据源的配置。以下是其“源”视图代码：

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="SqlDataSourceTest.Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:SqlDataSource ID="SqlDataSource1" runat="server"
                ConflictDetection="CompareAllValues"
                ConnectionString="<%= ConnectionStrings:
                AdventureWorks2008ConnectionString %>"
                DeleteCommand="DELETE FROM [DatabaseLog] WHERE
                    [DatabaseLogID] = @original_DatabaseLogID
                    AND [DatabaseUser] = @original_DatabaseUser
                    AND [Event] = @original_Event
                    AND [PostTime] = @original_PostTime"
                InsertCommand="INSERT INTO [DatabaseLog]
                    ([DatabaseUser], [Event], [PostTime])
                    VALUES (@DatabaseUser, @Event, @PostTime)"
                OldValuesParameterFormatString="original_{0}"
                SelectCommand="SELECT [DatabaseLogID], [DatabaseUser],
                    [Event], [PostTime] FROM [DatabaseLog]
                    ORDER BY [DatabaseLogID]"
                UpdateCommand="UPDATE [DatabaseLog]
                    SET [DatabaseUser] = @DatabaseUser,
                    [Event] = @Event, [PostTime] = @PostTime
                    WHERE [DatabaseLogID] = @original_DatabaseLogID
                    AND [DatabaseUser] = @original_DatabaseUser
                    AND [Event] = @original_Event
                    AND [PostTime] = @original_PostTime">
                <DeleteParameters>
                    <asp:Parameter Name="original_DatabaseLogID" Type="Int32" />
                    <asp:Parameter Name="original_DatabaseUser" Type="String" />
                    <asp:Parameter Name="original_Event" Type="String" />
                    <asp:Parameter Name="original_PostTime" Type="DateTime" />
                </DeleteParameters>
                <UpdateParameters>
                    <asp:Parameter Name="DatabaseUser" Type="String" />
                    <asp:Parameter Name="Event" Type="String" />
                    <asp:Parameter Name="PostTime" Type="DateTime" />
                    <asp:Parameter Name="original_DatabaseLogID" Type="Int32" />
                    <asp:Parameter Name="original_DatabaseUser" Type="String" />
                    <asp:Parameter Name="original_Event" Type="String" />
                    <asp:Parameter Name="original_PostTime" Type="DateTime" />
                </UpdateParameters>
                <InsertParameters>
                    <asp:Parameter Name="DatabaseUser" Type="String" />
                    <asp:Parameter Name="Event" Type="String" />
                    <asp:Parameter Name="PostTime" Type="DateTime" />
                </InsertParameters>
            </asp:SqlDataSource>

        </div>
    </form>
</body>
</html>
```

```
</body>
</html>
```

其中, `SqlDataSource` 对象的 `ConnectionString` 属性, 定义了基础数据库的连接字符串为 `AdventureWorks2008ConnectionString`。该字符串就是图 26.5 所示的对话框中所命名的字符串。该字符串被保存在 `Web.Config` 中的 `<configuration>` 节点里面。其定义如下:

```
<connectionStrings>
  <add name="AdventureWorks2008ConnectionString" connectionString="Data
    Source=.\SQLEXPRESS;Initial Catalog=AdventureWorks2008;
    Integrated Security=True" providerName="System.Data.SqlClient"/>
</connectionStrings>
```

而 `SelectCommand`、`DeleteCommand`、`InsertCommand` 和 `UpdateCommand` 分别定义了数据库的选择、删除、插入和修改命令。`<DeleteParameters>` 等标签, 表示其相关数据库命令属性所使用的参数集合。

在 26.2.3 节中将通过 `GridView` 控件来完成该数据源中数据的显示, 以及编辑等复杂功能。

26.2.3 用 GridView 控件显示数据

`GridView` 控件以表格的形式显示数据源中的数据。它提供了对列进行编辑列、添加列, 以及启用分页、启用排序、编辑记录、删除记录等功能。

需要说明的是, `GridView` 控件是 ASP.NET 早期版本中提供的 `DataGrid` 控件的后继控件。该控件不但可以利用 `SqlDataSource` 控件的新功能, 还做了一定的改进。下面将在 26.2.2 节介绍的例程基础上, 介绍有关 `GridView` 控件的排序、编辑记录、删除记录等功能。实现步骤如下所述。

- (1) 打开网站 `SqlDataSourceTest` 的 `Default.aspx` 页面, 并将页面切换至“设计”视图。
- (2) 在“工具箱”中的“数据”栏中找到 `GridView`, 并拖放至设计界面。
- (3) 单击“`GridView` 任务”中的“自动套用格式...”按钮, 弹出“自动套用格式”对话框, 任意选择其中一种格式(如专业型), 为 `GridView` 控件设计外观, 如图 26.10 所示。

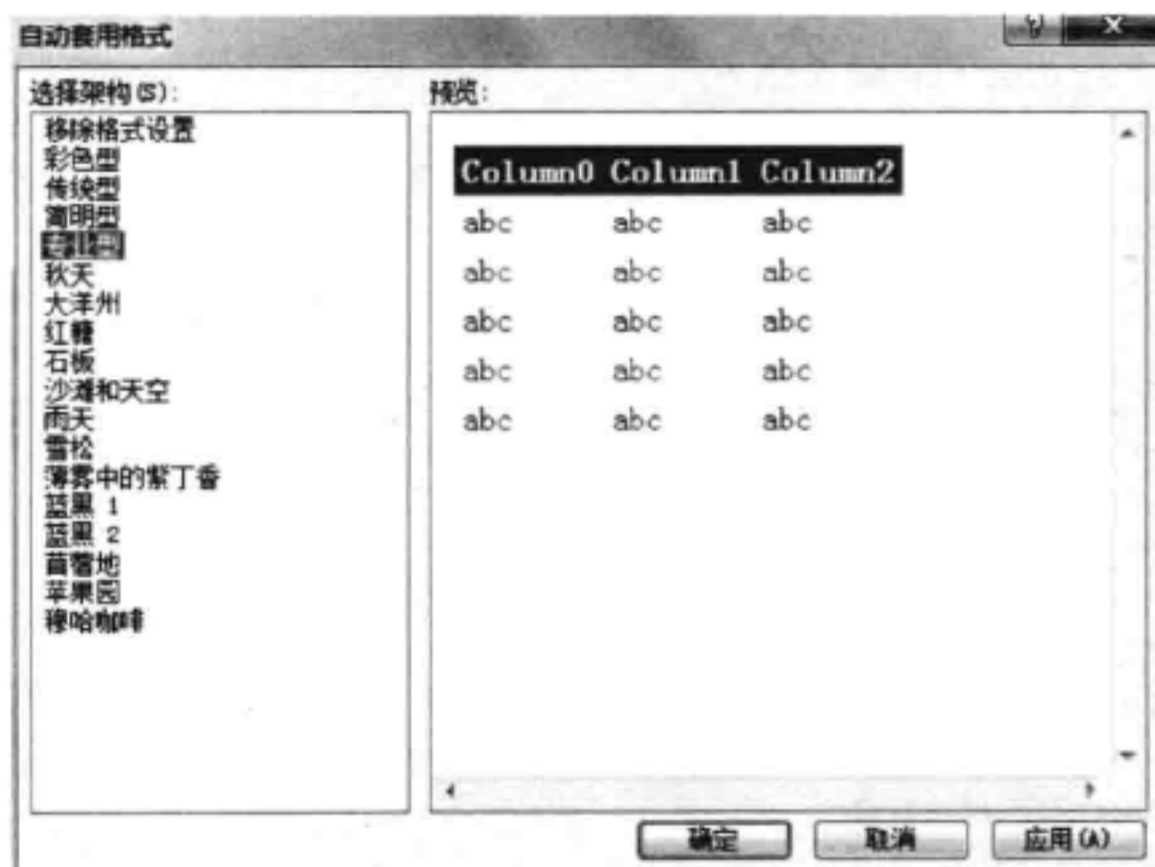


图 26.10 设计控件外观

(4) 单击“应用”和“确定”按钮，返回到原编辑界面，然后再单击 GridView 控件右上角的小箭头，重新打开“GridView 任务”，并为其选择建立的数据源 SqlDataSource1，将它们绑定在一起，如图 26.11 所示。



图 26.11 绑定数据源

(5) 将“GridView 任务”中的“启用分页”、“启用排序”、“启用编辑”、“启用删除”和“启用选定内容”都选中后，可以看到如图 26.12 所示的效果图。

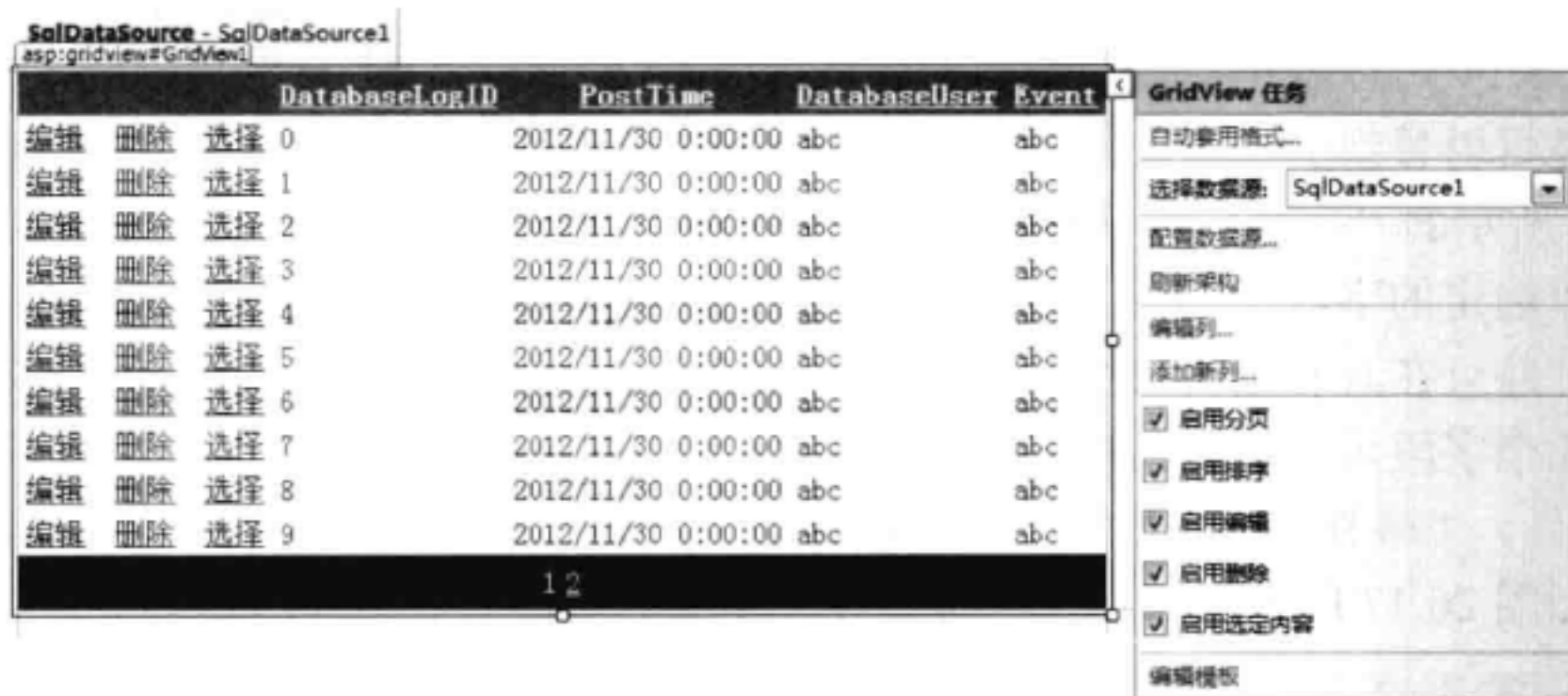


图 26.12 设计效果图

这样就实现了分页功能、编辑功能，以及删除、排序等功能。当然，还可以编辑某一行。

(6) 在 GridView 表格单击 Event 列标头，然后单击 GridView 控件右上角的小箭头，可以看到“GridView 任务”多出了用于移动列的显示位置的项，如“左移列”、“移除列”等，如图 26.13 所示。可以通过单击它们移动选定的列，以布局页面显示，此处单击“左移列”选项便可将 Event 列向左移动一列。

(7) 至此，就完成了数据源到 GridView 控件的显示了。运行程序，结果如图 26.14 所示。

此时就可以对记录进行编辑、删除和选择，并且可以通过单击每列的列标头以该列为依据进行排序。首先单击第一条记录对应的“编辑”链接，得到如图 26.15 所示的编辑状态的记录。在其中的 DatabaseUser 中，将 dbo 改为其他名称，如 wuyunhui。



图 26.13 布局列



图 26.14 程序运行界面

	DatabaseLogID	PostTime	Event	DatabaseUser
更新 取消	1	2011/3/28 22:43:48	CREATE_TABLE	dbo

图 26.15 修改记录

输入内容后，单击最左边的“更新”链接，完成对更新的保存。同样，也可以通过单击“删除”链接将所在行记录删除，并且这些更改都会自动提交到数据库进行更新。

从图 26.14 可以看出，显示数据的表格标头不是很好看，它只用了数据库中的列名。这里同样是可以替换的。可以打开“GridView 任务”菜单，单击“编辑列...”命令，得到如图 26.16 所示的“字段”对话框。

选择“选定的字段(S):”对应列表框中的 CommandField，为表中的命令字段添加表格标头，并且在其右边的属性列表框中找到其 HeaderText 属性，并输入“数据操作”。对于其他几个字段可以按照同样的方法修改。将它们的 HeaderText 分别命名为“记录标识”、“提交时间”、“事件”和“用户”。然后单击“确定”按钮，可以看到已经成功修改了标头内容，如图 26.17 所示。



图 26.16 添加和修改标头

数据操作	记录标识	提交时间	事件	用户
编辑 删除 选择	0	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	1	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	2	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	3	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	4	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	5	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	6	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	7	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	8	2012/11/30 0:00:00	abc	abc
编辑 删除 选择	9	2012/11/30 0:00:00	abc	abc

图 26.17 标头修改结果

26.2.4 用 DetailsView 控件显示指定记录

DetailsView 控件与前面介绍的 GridView 控件有许多相似之处，它们都用于数据的显示，只是 DetailsView 控件一次只显示一条表格中的记录，它也提供翻阅多条记录，以及插入、删除和更新记录的功能。DetailsView 控件通常用在“主要信息/详细信息”数据的现实方案中。在该种方案中，在主控件（如 GridView 控件）中选定的记录，决定了 DetailsView 控件中显示的记录内容。

下面在 26.2.2 节和 26.2.3 节介绍的例程基础上演示 DetailsView 控件的使用。主要是该控件与 GridView 控件的配合，其中包括了分页显示、记录的插入、删除和更新等操作。实现步骤如下所述。

（1）打开网站 SqlDataSourceTest 中的 Default.aspx，并将页面切换至“设计”视图。

（2）将 GridView 控件的“GridView 任务”中的“启用编辑”和“启用删除”两个复选框取消选定状态。

（3）在“工具箱”中的“数据”栏中找到 DetailsView 控件，拖放至设计界面，并展开“DetailsView 任务”，然后单击“自动套用格式...”按钮，如选择“沙滩和天空”格式，效果如图 26.18 所示。



图 26.18 套用格式

（4）单击“确定”按钮后，为 DetailsView 控件添加一个数据源，在“DetailsView 任务”中的“选择数据源：”对应的下拉列表框中选择“<新建数据源...>”选项，如图 26.19 所示。

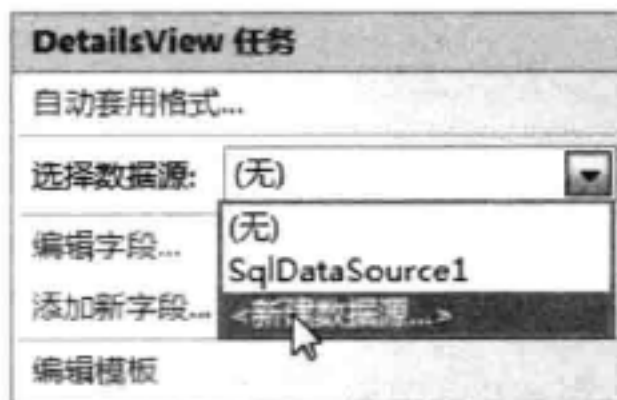


图 26.19 新建数据源

(5) 此处是通过“数据源配置向导”来添加数据源的。这时会弹出“数据源配置向导”对话框,如图 26.20 所示。在“应用程序从哪里获取数据(W)?”列表框中选择“数据库”,然后为数据源指定 ID,此处保持其默认名 SqlDataSource2。



图 26.20 选择数据源类型

(6) 单击“确定”按钮后,弹出“选择您的数据连接”对话框,如图 26.21 所示。由于前面已经建立了连接到数据库 AdventureWorks2008 的连接字符串 AdventureWorks2008-ConnectionString, 因此可以在下拉列表框中找到该连接字符串, 并选择它。



图 26.21 配置数据库连接字符串

(7) 这时单击“下一步”按钮,弹出“配置 Select 语句”对话框。由于本例程中主表 GridView 中显示的是数据库 AdventureWorks2008 中的 DatabaseLog 表中的数据,因此选择“名称”下拉列表框中的 DatabaseLog, 然后勾选“列”列表框中的需要 DetailsView 显示

的详细信息，本例程勾选项如图 26.22 所示。



图 26.22 配置 Select 语句

(8) 单击“WHERE (W) ...”按钮，为 SQL 语句添加查询条件，将弹出“添加 WHERE 子句”对话框，如图 26.23 所示。

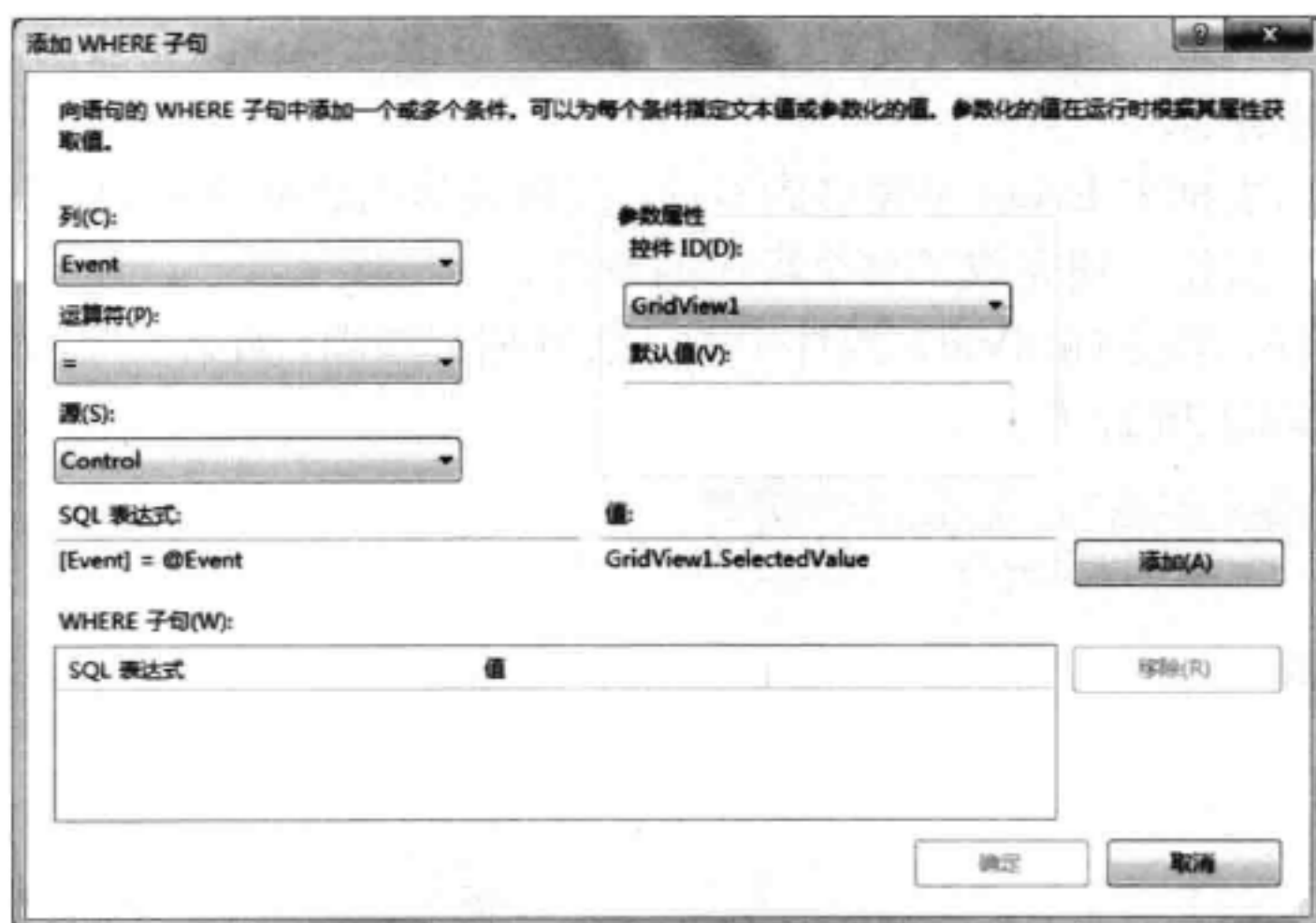


图 26.23 添加 WHERE 子句

为了便于说明控件的分页功能，选择其中的“列”下拉列表框中的 Event 选项，选择“源”下拉列表框中的 Control 选项，表示来自页面上的控件。同时选择其右边“参数属性”组框中的“控件 ID(D):”为 GridView1，表示与 DetailsView 控件关联的控件为 GridView1。然后单击“添加”按钮和“确定”按钮，返回到“配置 Select 语句”对话框，可以看到这时的“SELECT 语句”变为了如下语句：

```
SELECT [DatabaseLogID], [PostTime], [DatabaseUser], [Event], [Schema],
[Object] FROM [DatabaseLog] WHERE ([Event] = @Event)
```

(9) 单击该页面中的“高级(V)…”按钮,得到如图 26.24 所示对话框,勾选第一个复选框,以便使 DetailsView 具有对数据库记录的增、删、改功能。



图 26.24 高级 SQL 生成选项

(10) 单击“确定”按钮,然后单击“下一步”按钮,最后单击“完成”按钮,完成对 SqlDataSource2 的配置。

(11) 打开“DetailsView 任务”,这时菜单中出现了“启用分页”、“启用插入”、“启用编辑”和“启用删除”复选项,勾选这些复选项,使之具有相应的功能。

(12) 选中 GridView 控件,为其配置键字段,在其“属性”框中找到 DataKeyNames,可以看到其默认属性值为 DatabaseLogID。由于在数据源 SqlDataSource2 中配置的 WHERE 条件中的字段是 Event,因此,此处也应该为相应的字段。单击其右边的浏览按钮,得到如图 26.25 所示对话框。

将其中左边列表框中 Event 字段移到右边,而将右边列表框中的 DatabaseLogID 移出,然后单击“确定”按钮,便完成了整个程序的设计。

(13) 运行程序,单击 GridView 控件中的任意数据记录项,如第 5 项,得到 DetailsView 的运行效果图,如图 26.26 所示。



图 26.25 配置查询依据的字段

DatabaseLogID	5
PostTime	2011/3/28 22:43:49
DatabaseUser	dbo
Event	CREATE_TYPE
Schema	dbo
Object	AccountNumber
编辑 删除 新建	
	123456

图 26.26 DetailsView 的分页效果

可以看出,所有字段 Event 值为 CREATE_TYPE 的记录都被选出来了,总共 6 条记录,并分 6 页显示。单击“编辑”链接,可以进入编辑页面,如图 26.27 所示。

最后通过“更新”按钮来保存编辑的修改。“删除”链接表示删除当前选定的记录行。“新建”链接用于插入一条记录到数据库。需要说明的是,新建记录时要求非空的字段必须保证有值,不然会导致插入失败,其界面和图 26.27 类似,此处不再赘述。

DatabaseLogID	5
PostTime	2011/3/28 22:43:49
DatabaseUser	dbo
Event	CREATE_TYPE
Schema	dbo
Object	AccountNumber
更新 取消	
123456	

图 26.27 编辑选定的记录

26.3 用验证控件检查用户输入

用户在网页的控件上面输入数据，往往需要对其进行有效性验证。这样可以及时检查用户提交的数据是否合法，同时可以减少服务器的负担。

26.3.1 必填内容的验证控件 RequiredFieldValidator

它用于限制空字段，在页面提交前不允许输入框为空。在工具箱中的“验证”一栏中可以找到这个控件，如图 26.28 所示。下面将通过实例介绍该控件的用法，步骤如下所述。

(1) 打开 Visual Studio 2010，新建一个 ASP.NET 空 Web 应用程序，并命名为 ValidatorTest。

(2) 将页面切换到“设计”视图，并添加一个具有 4 行 3 列的表。

(3) 在表中拖放 3 个 Label 控件、3 个 TextBox 控件和一个 Button 控件。为 3 个 Label 和 Button 控件添加 Text 属性，并布局界面，如图 26.29 所示。



图 26.28 验证控件

用户名：	<input type="text"/>	
密码：	<input type="password"/>	
年龄：	<input type="text"/>	
		提交

图 26.29 布局控件

(4) 分别为 3 个 TextBox 控件设置 ID 属性。其中“用户名”对应为 UserNameTxt，“密码”对应为 PassWordTxt，“年龄”对应为 AgeTxt。

(5) 在“工具箱”中的“验证”栏找到 RequiredFieldValidator 控件，并拖放至用户名输入文本框对应的右边表格中。

(6) 选定该控件，在其属性窗口中找到 ErrorMessage，并将其设置为“*必需填写用户名”。同时，将 ControlToValidate 属性值设置为 UserNameTxt。

这两个属性，前者用于设置验证错误发生时要显示在控件上的信息，后者用于将当前控件与 ID 名为 UserNameTxt 的控件关联。

运行程序，测试 RequiredFieldValidator 控件的效果。如果“用户名”中不输入任何信息就单击“提交”按钮，该控件上会显示 ErrorMessage 里设置的信息，即“*必需填写用户名”，如图 26.30 所示。



图 26.30 非空检查

说明：RequiredFieldValidator 控件还有一个比较重要的属性就是 display，它有 3 个可能的取值，分别是 None、Static 和 Dynamic。表 26.1 列出了它们的详细解释。

表 26.1 display 的属性值及其意义

属 性 值	意 义
Dynamic	动态显示验证错误信息，其在页面上的位置空间只有在验证错误发生时才给出
None	验证错误信息，但是不显示到页面上
Static	与Dynamic不同，该属性使得不管验证错误是否发生，页面都会留出供显示错误信息的空间

由于属性值为 Dynamic 时，验证错误信息是动态给出的。因此，如果之前该空间被其他页面内容占据时，会导致这部分内容的显示位置发生变化，从而可能使页面布局并非预想的那样。所以在选择属性值的时候，应该在兼顾布局的基础上，充分利用空间，综合利用这 3 种取值的优势。

26.3.2 比较两个值的验证控件 CompareValidator

可以使用 CompareValidator 与固定值比较，也可以对两个控件进行比较，也可以用于检查数据类型。

首先是 CompareValidator 与固定值的比较。为此还是利用前面介绍的例程说明该用法，设计步骤如下所述。

(1) 打开网站 ValidatorTest 中的 Default.aspx，将页面切换到“设计”视图，然后在年龄对应的文本框的右边表格里添加两个验证控件，一个是 RequiredFieldValidator 控件，另一个是 CompareValidator 控件。前者用于验证输入非空，后者用于比较输入的年龄数字是否与给定的数字相符。

(2) 设置 RequiredFieldValidator 控件的属性，如表 26.2 所示。

表 26.2 设置RequiredFieldValidator的属性

属 性	属 性 值	属 性	属 性 值
ErrorMessage	*年龄不能为空	Display	Dinamic

然后设置 CompareValidator 控件的属性如表 26.3 所示。

表 26.3 设置CompareValidator的属性

属 性	属 性 值	属 性	属 性 值
ErrorMessage	*年龄不能小于20岁	ValueToCompare	20
Display	Dynamic	Type	Integer
Operator	GreaterThanOrEqual		

将两个控件的 Display 属性都设置为 Dynamic, 是为了显示验证错误时的布局紧凑和美观。CompareValidator 的 Operator 设置为 GreaterThanOrEqual 表示大于等于, 用于验证控件中的输入是否大于等于 ValueToCompare 属性给定的值。Type 属性指定了要比较的值(此处是 20)的数据类型, 它还可以是 String、Double、Date 和 Currency 等类型。其中 Date 表示日期类型, Currency 是货币类型。

(3) 运行程序, 测试上面对控件设置的效果, 如图 26.31 所示。当输入的年龄小于 20 时则会显示验证信息, 如图 26.32 所示。



图 26.31 非空检查



图 26.32 比较验证

可以看出两个验证控件所显示的位置都是一样的, 这是 Display 的 Dynamic 属性值在起作用。当输入的年龄大于或等于 20 时, 可以通过验证。

接下来是使用 CompareValidator 控件对两个控件的输入进行比较。下面将对“用户名”和“密码”是否一致进行比较, 设计步骤如下所述。

(1) 打开网站 ValidatorTest 中的 Default.aspx, 将页面切换到“设计”视图。在“工具箱”中找到 CompareValidator 控件, 并拖放至页面中“密码”对应的输入框右边的表格中。

(2) 为 CompareValidator 控件设置属性, 如表 26.4 所示。

表 26.4 设置CompareValidator的属性

属 性	属 性 值	属 性	属 性 值
ErrorMessage	密码和用户名不能相同	ControlToCompare	UserNameTxt
Display	Static	Type	String
Operator	NotEqual	TextModel	Password
ControlToValidate	PassWordTxt		

(3) 上述设置就实现了两个输入控件内容的比较。Operator 设置为 NotEqual, 表示两个控件内容不等时通过验证。ControlToValidate 表示用于验证的控件, 此处为 PassWordTxt 控件。ControlToCompare 为 UserNameTxt, 表示用于比较的另外一个控件 ID。TextModel 表示输入的文本以密码的形式显示。

(4) 运行程序可以看到, 当输入的用户名和密码不同时, 验证通过。当输入的用户名和密码相同时, 验证不能通过, 并且显示验证的提示信息, 如图 26.33 所示。



图 26.33 验证两个控件的输入内容

最后是用户数据类型检查, 其设计步骤如下所述。

(1) 打开网站 ValidatorTest 中的 Default.aspx, 将页面切换到“设计”视图。

(2) 在表中添加一行, 为此鼠标选定表的最后一行。右键单击该行, 在弹出的快捷菜单中选择“插入”|“上面的行(A)”命令。

(3) 在新添加的行中添加一个 Label 控件、一个 TextBox 控件和一个 CompareValidator 控件, 并将 Label 控件的 Text 属性设置为“出生年月”。将 CompareValidator 控件的 ID 属性设置为 AgeTxt。

(4) 将 CompareValidator 控件的属性设置为如表 26.5 所示。

表 26.5 设置 CompareValidator 的属性

属 性	属 性 值	属 性	属 性 值
ErrorMessage	日期格式错误	ControlToValidate	BirthdateTxt
Display	Dynamic	Type	Date
Operator	DataTypeCheck		

Operator 的属性设置为 DataTypeCheck, 表示进行日期类型检查操作。Type 设置为 Date 表示比较的值的数据类型为日期类型。

(5) 运行程序, 得到如图 26.34 所示的效果。如果输入的日期超出实际表示范围, 这时会发生日期格式错误信息提示, 如图 26.35 所示。



图 26.34 日期类型检查



图 26.35 日期格式错误

26.3.3 检查指定范围的验证控件 RangeValidator

该控件用于检查用户输入数据的范围。这种数据可以是数字、字符串、日期等。用户可以通过 RangeValidator 控件的属性来指定数据的范围。下面将以前面介绍的例子为例，介绍 RangeValidator 控件的使用方法，设计步骤如下所述。

(1) 打开网站 ValidatorTest 中的 Default.aspx，将页面切换到“设计”视图。在“工具箱”中找到 RangeValidator 控件，并拖放至页面中“出生年月”对应的输入框右边的表格中（其中已经有一个 CompareValidator 控件）。

(2) 设置 RangeValidator 控件的属性如表 26.6 所示。

表 26.6 设置RangeValidator的属性

属 性	属 性 值	属 性	属 性 值
ErrorMessage	出生日期超出指定范围	MaximumValue	2027-9-1
Display	Dynamic	MinimumValue	1997-9-1
ControlToValidate	BirthdateTxt	Type	Date

(3) 其中 MaximumValue 和 MinimumValue 指定了控件中允许输入的范围。此处表示日期范围。

(4) 运行程序可以看到，当输入的日期范围超出指定的日期范围时，范围验证控件会显示验证错误信息，如图 26.36 所示。

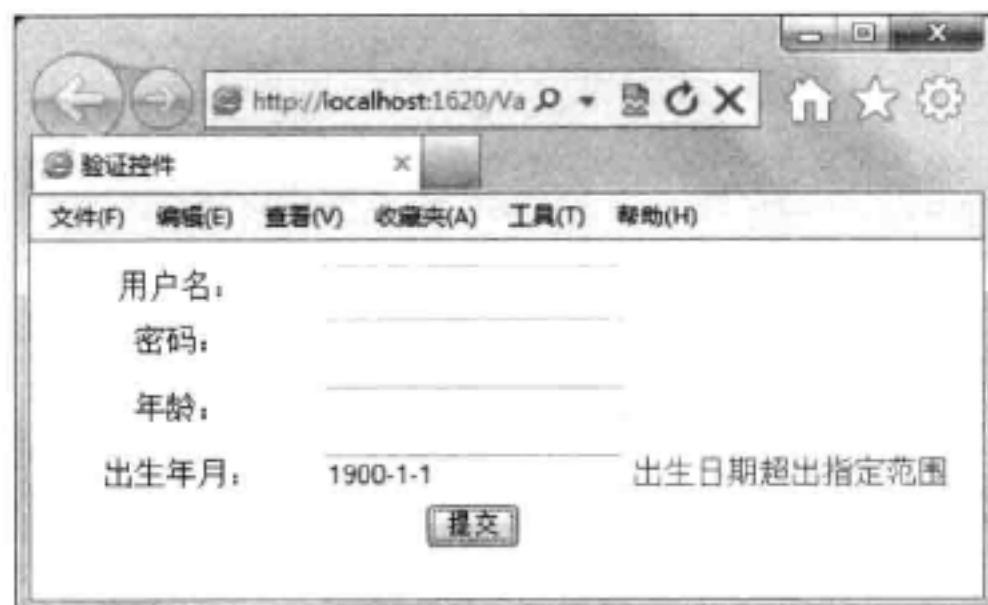


图 26.36 范围验证控件

26.3.4 正则表达式验证控件 RegularExpressionValidator

网页中的输入控件有时需要帮助用户对输入数据做简单的格式判断，如电子邮件地址、电话号码、身份证、邮政编码等。这些数据都有一个共同点，就是它们都具有一些相对固定的格式。如电子邮件地址中都有一个“@”符号，国内身份证号为 15 位或 18 位。ASP.NET 提供了对应的表达式验证控件 RegularExpressionValidator，用于验证这些输入是否符合预先定义好的格式。定义的这个格式就是正则表达式。

下面将以前面介绍的例子为基础，介绍表达式验证控件 RegularExpressionValidator 的用法，设计步骤如下所述。

(1) 打开网站 ValidatorTest 中的 Default.aspx, 将页面切换到“设计”视图。在表中按照前面介绍的方法, 添加一行到表格中, 并在工具箱中拖放一个 Label 控件、一个 TextBox 控件和一个 RegularExpressionValidator 控件。

(2) 将 Label 控件的 Text 属性设置为“电子邮件地址:”, 将 TextBox 控件的 ID 属性设置为 EmailTxt。

(3) 设置 RegularExpressionValidator 控件属性, 如表 26.7 所示。

表 26.7 设置 RangeValidator 的属性

属 性	属 性 值	属 性	属 性 值
ErrorMessage	电子邮件格式错误	ControlToValidate	EmailTxt
Display	Static	ValidationExpression	\w+([-+.'\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*

其中的 ValidationExpression 是用于确定输入格式的正则表达式。读者如果不熟悉该表达式也不必担心, 此处可以通过 ValidationExpression 属性提供的“正则表达式编辑器”来完成。为此, 单击该属性项中提供的“浏览”按钮, 得到如图 26.37 所示“正则表达式编辑器”对话框。

其中还提供了网络 URL 表达式、各国电话号码表达式、邮政编码、电话号码等常用的验证表达式。此处选择“Internet 电子邮件地址”选项, 然后单击“确定”按钮, 便完成了电子邮件地址的正则表达式设置。

(4) 运行程序, 效果如图 26.38 所示。

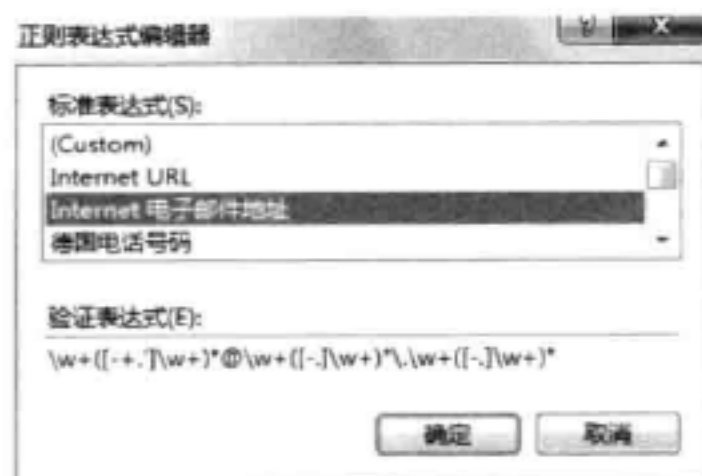


图 26.37 “正则表达式编辑器”对话框



图 26.38 表达式验证控件

26.3.5 自定义验证控件 CustomValidator

自定义验证控件用于解决在数据验证时, 验证方式的个性需求。前面介绍的几个控件都属于客户端验证控件, 也只能用于客户端验证。而自定义控件 CustomValidator, 则可以实现服务器端的数据验证, 如数据库中的数据对比等。

下面将介绍如何使用 CustomValidator 控件, 验证用户所提交的用户名在服务器端数据库中是否已经存在。为此首先需要建立一张用于存储用户名的表, 设计步骤如下所述。

(1) 打开第 23 章中建立的 mydb 数据库, 在其中建立一个用于存储用户名的表 UserName, 并添加列名及其数据类型, 如图 26.39 所示。

(2) 在表中添加一条记录 aaa，如图 26.40 所示。



图 26.39 建立用户表

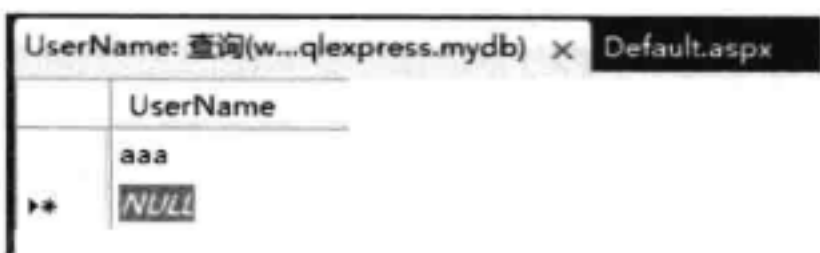


图 26.40 添加记录

(3) 完成表的建立之后，就可以编写数据验证的代码了。打开网站 ValidatorTest 中的 Default.aspx，将页面切换到“设计”视图。

(4) 在“用户名”对应的输入框右边的表格中，再拖放一个 CustomValidator 用户自定义控件，并设置其属性如表 26.8 所示。

表 26.8 设置CustomValidator的属性

属 性	属 性 值	属 性	属 性 值
ErrorMessage	该用户名已经存在	ControlToValidate	UserNameTxt
Display	Dynamic		

(5) 由于需要在服务器端进行数据验证，此处将用到服务器端验证事件 ServerValidate。为此，选定 CustomValidator 控件，在“属性”窗口中的事件栏中找到 ServerValidate。双击该项，为其编写数据验证事件处理程序，代码如下：

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Data.SqlClient;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void CustomValidator1_ServerValidate(object source, Server
        ValidateEventArgs args)
    {
        //实例化一个数据库连接
        SqlConnection con = new SqlConnection();
        //设置数据库连接字符串
        con.ConnectionString = "Data Source=.\SQLEXPRESS;"
            + "Integrated Security=SSPI;Initial Catalog=mydb";
        //打开数据库
        con.Open();
        SqlCommand cmd = new SqlCommand();
        //执行数据库查询
        cmd.CommandText = "select count(*) from UserName where UserName='"
            + args.Value + "'";
        cmd.Connection = con;
```

```

//数据类型转换
int result = Convert.ToInt32(cmd.ExecuteScalar());
//指定控件是否通过验证
if (result > 0)
{
    args.IsValid = false;
}
else
{
    args.IsValid = true;
}
}
}

```

程序首先引入命名空间 `System.Data.SqlClient`，以调用操作数据库相关类。然后在 `ServerValidate` 的事件处理程序 `CustomValidator1_ServerValidate` 中编写处理代码。

前半部分代码用于打开数据库、查找数据库记录数，并把记录数返回给变量 `result`。

下面这段代码用到了 `ServerValidateEventArgs` 的对象 `args`。这是通过事件处理程序的参数传递下来的，用于传递事件数据。

```

if (result > 0)
{
    args.IsValid = false;
}
else
{
    args.IsValid = true;
}
}

```

其中的属性 `Value` 用于获取要验证的值，这个值就是用户输入文本框中的值，因此也可以使用常规的 `this.UserNameTxt.Text` 方法直接获取。而属性 `IsValid` 表示控件是否通过验证。在程序中，当 `result` 大于 0 时表示数据库中已经有相同的记录了，因此将 `IsValid` 设置为 `false`，使控件发生验证错误；反之则设置为 `true`，使控件通过验证。运行程序，并输入数据，结果如图 26.41 所示。

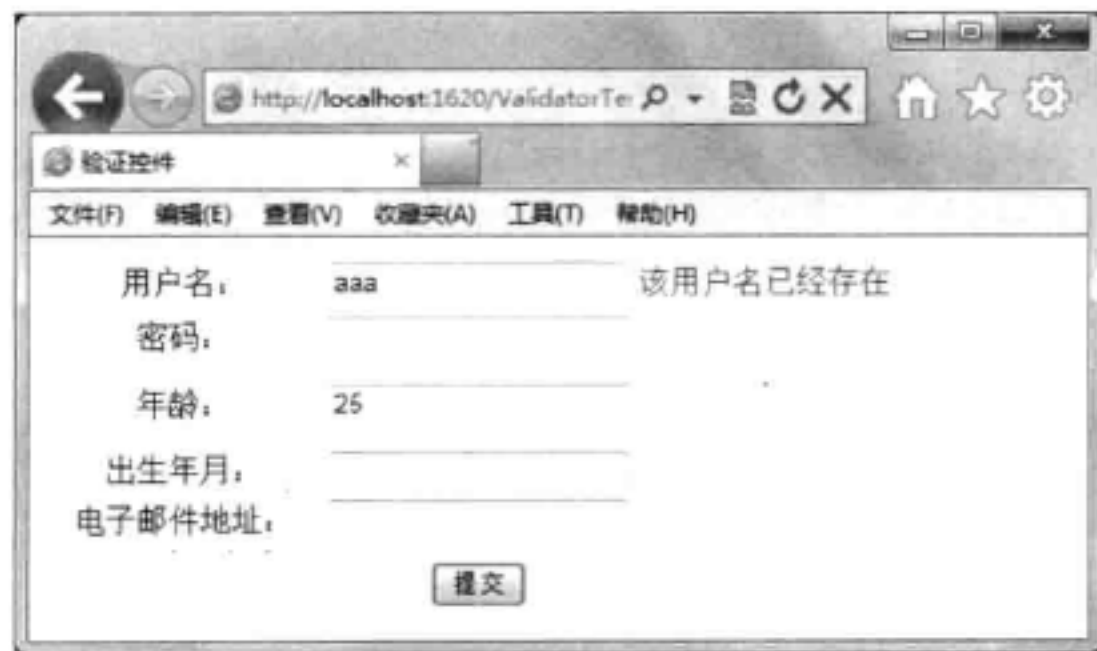


图 26.41 自定义验证结果

由于表中没有 `www` 这样的记录，因此单击“提交”按钮是不会出错的。如果输入 `aaa` 这条记录，由于数据库中已经存放了该记录，单击“提交”按钮后，就会发生验证错误。

当然，也可以通过在客户端编写脚本，实现 `CustomValidator` 控件的自定义验证。此时则需通过其属性 `ClientValidationFunction` 来设置验证脚本函数。

26.3.6 验证错误信息汇总控件 ValidationSummary

错误信息汇总控件用于将页面中所有验证控件错误信息显示出来。如果验证控件分为不同的组时，则需要不同的 ValidationSummary 控件进行分组显示。默认情况下，页面控件都在一个组中。设置分组信息用到了 ValidationGroup。ValidationSummary 控件用法比较简单，下面将以前面的例子对其应用做简单的介绍，设计步骤如下所述。

(1) 打开网站 ValidatorTest 中的 Default.aspx，将页面切换到“设计”视图。在“提交”按钮所在行的表格中添加一个 ValidationSummary 控件。

(2) 保持其属性为默认值，调整好布局，并运行程序，结果如图 26.42 所示。

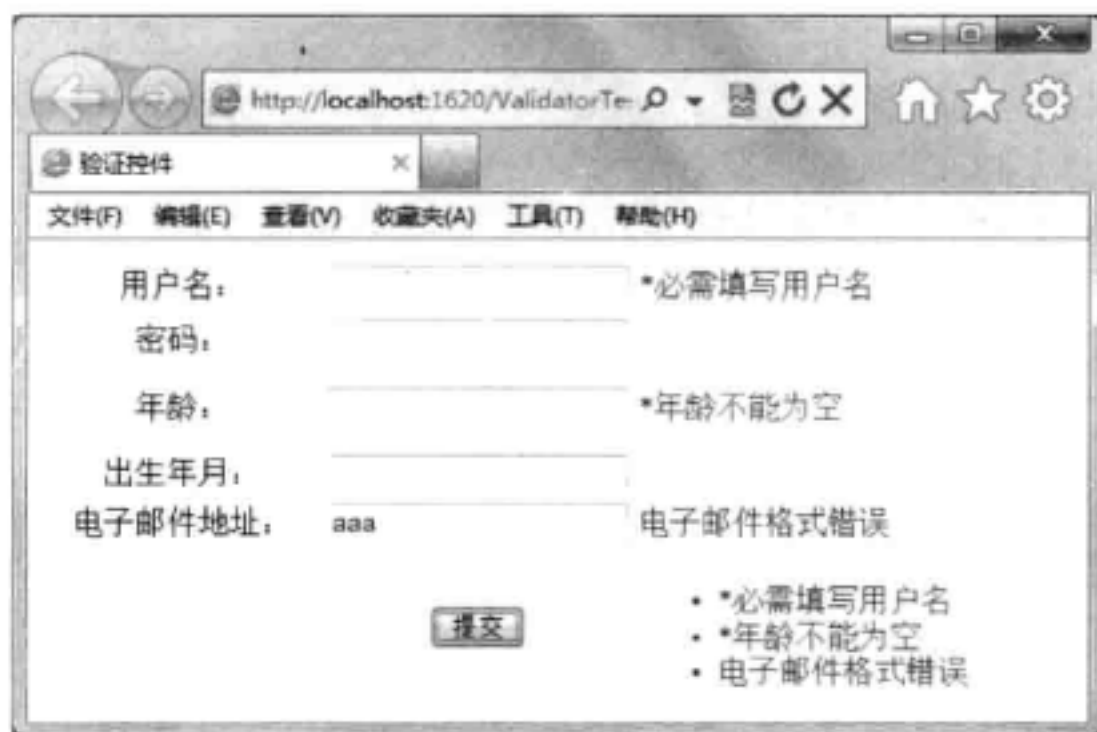


图 26.42 错误信息汇总

单击“提交”按钮后可以看到，错误信息汇总控件 ValidationSummary 将页面中所有控件的错误信息都显示出来了。这时可以将页面上的其他控件的 Display 属性设置为 None，即不显示出来，而直接汇总显示到 ValidationSummary 控件中，从而可以使页面布局更加紧凑。

最后需要说明的是一个有关页面验证的属性，这就是 IsValid，它是 Page 类中的一个属性。该属性用于指示整个页面是否全部验证通过。如果验证没有通过，则不提交到服务器端执行。当然，即使页面中有需要在服务器端验证的控件，如前面介绍的用户自定义验证控件 CustomValidator，通过 IsValid 也可以判断整个页面验证是否全部通过。下面将介绍 IsValid 的使用方法。

在程序页面中拖放一个 Label 控件，并将其 ID 设置为 ResultLbl，用于显示整个页面的验证状态。双击“提交”按钮，为其编写事件处理程序，如下所示。

```
protected void Button1_Click(object sender, EventArgs e)
{
    //判断页面验证是否全部通过
    if (this.IsValid)
    {
        ResultLbl.Text = "页面验证成功";
    }
    else
    {
        ResultLbl.Text = "页面验证未成功";
    }
}
```

运行程序，可以看到效果如图 26.43 所示。

当所有控件验证通过时，可以看到如图 26.44 所示的运行结果。



图 26.43 验证未通过



图 26.44 验证通过

26.4 创建 ASP.NET 用户控件

虽然在 .NET 中已经提供了相当多的控件用于编写应用程序。但是，有时用户需要根据自己的需要，来编写和系统提供的控件具有类似效果（可以拖放、重用等功能）的控件，这就是用户自定义控件，在 ASP.NET 中提供了定义用户自定义控件的功能。下面的实例将简单介绍用户自定义控件的定义方法。在很多网站中，为了保持页面的连续感，经常将页面的头部和尾部的内容和风格设置为相同或者相似的界面。下面将使用用户自定义控件的方法，建立一个网页的头部，设计步骤如下所述。

(1) 打开 Visual Studio 2010，新建一个 ASP.NET 空 Web 应用程序，并命名为 UserControlTest。

(2) 右键单击“解决方案资源管理器”中的 UserControlTest 项目，选择“添加新项”命令打开“添加新项”对话框，然后选择模板窗口中的“Web 用户控件”，并命名为 Header.ascx，如图 26.45 所示。



图 26.45 添加用户自定义控件

(3) 单击“添加”按钮，得到 Header.ascx 文件的“源”视图，其中的代码如下：

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="Header.ascx.cs"
Inherits="WebUserController" %>
```

与.aspx 页面不同，其中的第 1 行不是 Page 指令，而是 Control 指令。CodeFile 表示隐藏文件代码，放置在 Header.ascx.cs 文件中，这与.aspx 文件是对应的。Inherits 指定了其继续的类。

(4) 在页面中添加一个 3 行 6 列的表，并合并最后一行，在其中放置一个 HTML 控件 Horizontal Rule。

(5) 在表的第 1 和第 2 行中的表格中输入一些文字，作为网页导航，如图 26.46 所示。

新闻	体育	娱乐	明星	汽车	财经
科技	校园	生活	感情	资源	数码

图 26.46 添加导航栏

为了设置表格中文字的各种属性，可以通过如下方式进行设置。

首先，选定要设置对齐方式的所有文字或者表格，然后在“属性”窗口中找到 Style 属性，并单击其中的“浏览”按钮，得到如图 26.47 所示“修改样式”对话框。



图 26.47 设置字体样式

其中类别列表框中的字体、块、背景等列表项对应了文字的显示样式，同时下面给出了相应的效果预览和说明。然后设置一种样式，单击“确定”按钮，完成样式设置。

以下是该用户自定义控件中的所有源代码。

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="Header.ascx.cs"
Inherits="WebUserController" %>
<div style="text-align: center">
    <table>
        <tr>
            <td style="width: 100px; text-align: center;">
                <a href=Default.aspx>新闻</a></td>
            <td style="width: 100px; text-align: center;">
```

```

        <a href="Default.aspx">体育</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">娱乐</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">明星</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">汽车</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">财经</a></td>
    </tr>
    <tr>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">科技</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">校园</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">生活</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">感情</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">资源</a></td>
        <td style="width: 100px; text-align: center;">
            <a href="Default.aspx">数码</a></td>
    </tr>
    <tr>
        <td colspan="6">
            <hr />
        </td>
    </tr>
</table>
</div>

```

可以看到该代码中是没有<html>、<head>、<body>和<form>标签的，这些标签是不允许存在于用户自定义控件中的。因为用户控件只是一个网页的片断，不具备完整的网页格式，因此用户自定义控件必须放置到已有的可执行页面中才能运行。

(6) 接下来便是在主页面中引用该用户自定义控件（添加一个名为 Default.aspx 的主页面）。为此，最简单的方法就是直接将其从“解决方案资源管理器”中拖放到页面即可。如图 26.48 所示为其拖放效果。

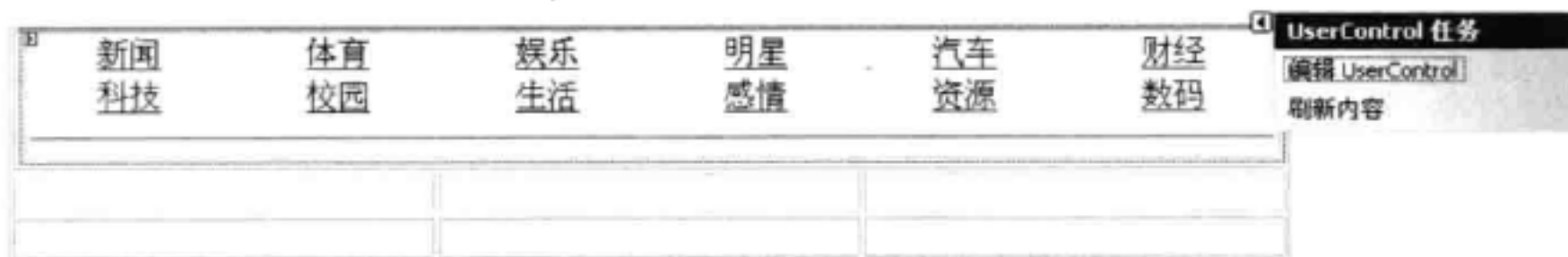


图 26.48 引用用户自定义控件

当然，也可以在“UserControl 任务”中编辑原用户自定义控件中的内容，或者在用户控件中添加其他外部控件。同时也可以对用户控件进行编程。这样，就可以在整个网站的所有页面都放置 Header.ascx 控件，从而极大提高了代码的重用性。

最后需要说明的是，在引用用户自定义控件的页面中，其源代码用到了如下语句：

```
<%@ Register Src="Header.ascx" TagName="Header" TagPrefix="uc1" %>
```

其中，Register 表示在该页面注册该用户自定义控件。Src 代表用户控件的虚拟路径。

TagName 表示用户控件的名称。TagPrefix 代表定义用户控件的命名空间，它和用户控件的名称一起确定页面上的用户控件。

26.5 本章总结

本章主要介绍了 ASP.NET 中常用的 Web 控件，这些控件包括数据操作控件、验证控件，以及用户控件。其中数据操作控件用于完成数据库中数据查询、增加、删除、修改，以及数据的显示等功能。验证控件是用的比较多的一类控件，它极大地提高了程序员的开发效率，并且确保了数据格式、数据类型、数据范围等的正确性。同时它也提供了根据用户需要的灵活的验证功能，这就是自定义验证控件。

本章最后介绍了用户控件，这种控件可以根据需要实现代码的高度重用，从而方便了程序的开发和维护。

26.6 实战练习

1. 在 Visual Studio 2010 中新建一个 ASP.NET 空 Web 应用程序，添加一个 Default.aspx 页面，在该页面中用 GridView 控件显示 AdventureWorks2008 数据库中 Product 表中的数据。

2. 接第 1 题，修改 GridView 控件中的内容，使其只显示 ProductID 和 Name 数据，当用户在 GridView 控件中选中某一行数据时，在下方的 DetailsView 控件中显示 Product 的详细信息。

3. 在 Visual Studio 2010 中新建一个 ASP.NET 空 Web 应用程序，添加一个 Default.aspx 页面，设计一个用户注册界面，包含用户名、密码、验证密码、电子邮件等几项，为该页面的各控件设置验证控件，要求两次输入的密码必须相同，输入的电子邮件是合法有效的。

4. 在 Visual Studio 2010 中新建一个 ASP.NET 空 Web 应用程序，添加一个 Default.aspx 页面，在其中添加一个 TextBox 控件，用来验证在 AdventureWorks2008 数据库 Product 表中是否存在输入的产品名称。要求通过自定义验证控件来完成本题。

第 7 篇 .NET 4.0 的增强

功能

- ▶▶ 第 27 章 WPF 框架
- ▶▶ 第 28 章 WCF 框架
- ▶▶ 第 29 章 Windows WF 框架
- ▶▶ 第 30 章 语言集成查询 LINQ

第 27 章 WPF 框架

WPF 是 Windows Presentation Foundation 的缩写，它是微软公司发布的下一代显示系统框架，通过其强大的展现功能，使得用户体验更加丰富。基于 WPF 框架，开发人员可以创建强大的独立应用程序和基于浏览器的网页应用程序。

27.1 WPF 基础

WPF 提供了展现系统的架构，它的核心是提供了一个与分辨率无关，并且基于向量的呈现引擎。由于该框架内置了很多功能强大的应用程序开发组件，使 WPF 在利用现代图形的优势方面更加明显。这些组件包括可扩展应用程序标记语言（XAML）、控件、数据绑定、布局、图形、动画、样式、模板、文档、媒体、文本和版式等。

27.1.1 了解 WPF 基础架构

WPF 的基础架构是在原有的公共语言运行时（CLR）基础上，经过进一步的优化和补充实现了强大的展现功能，同时它也不局限于托管代码，通过与硬件加速的完美结合，实现了高效的硬件和软件呈现。WPF 的基础架构如图 27.1 所示。

图中，深色部分的模块是 WPF 的主要代码模块。在这些模块中，只有 Milcore 属于非托管组件，之所以使用非托管代码实现，目的是实现与 DirectX 的紧密集成。WPF 中的所有显示是通过 DirectX 引擎完成的，可实现高效的硬件和软件呈现。WPF 还要求对内存和执行进行精确控制，Milcore 中的组合引擎受性能影响较大，需要放弃 CLR 的许多优点来提高性能。图中的浅色部分为 .NET 框架本身提供的组件，用于配合 WPF 完成其展现功能。

WPF 旨在创建动态的数据驱动演示系统。系统的每一部分均可通过驱动行为的属性集来创建对象。数据绑定是系统的基础部分，在每一层中均进行了集成。传统的应用程序先创建一个显示内容，然后绑定到某些数据。在 WPF 中，关于控件的所有内容，以及显示内容的所有方面都是由某种类型的数据绑定生成的。通过在按钮内部创建复合控件并将其显示绑定到按钮的内容属性，就会显示按钮内的文本。

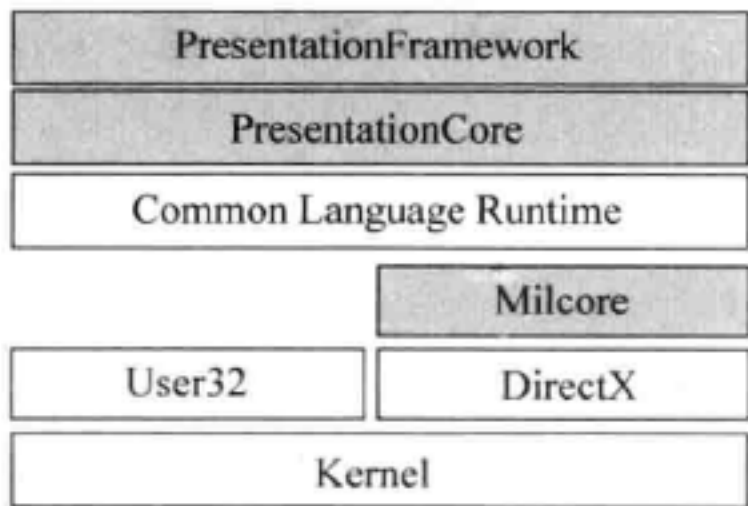


图 27.1 WPF 基础架构

27.1.2 与 WPF 相关的技术

WPF 是一个展现系统的框架总称，其架构下包含了很多具体的实现技术，这些技术包括以下几个方面。

- ❑ **XAML 语言：**XAML 语言是一种基于 XML 的标记语言，用于以声明的方式实现应用程序的外观。它通常用于创建窗口、对话框、页面和用户控件，并用控件、形状和图形填充它们。
- ❑ **控件：**WPF 控件是实现界面展现的基础元素，其适用于窗口程序或者页面程序承载。通过各种控件的组合，来实现与用户交互的用户展现窗口。事实上，控件是一组封装好的并且用于实现某些行为的 WPF 类。WPF 控件分为如下几个大类，分别是按钮、输入、菜单、布局、对话框、选择、数字墨迹、文档、媒体、导航、用户信息控件等。
- ❑ **布局：**WPF 窗口程序或者网页程序通过控件组成之后，需要通过一定的排列方式来显示出页面的层次或者区域特性，这个特性需要布局的功能来实现。实际上，布局的实现是通过布局控件来完成的，通用的布局包括网格、堆叠和停靠布局等。
- ❑ **数据绑定：**窗口程序很多场景需要提供数据的显示，以供用户进行阅读和操作。数据绑定完成了窗口控件与后台数据源之间的数据交互的过程。WPF 可以通过 ADO.NET 等相关技术，实现对数据的读取，将获取的数据绑定到对应的数据控件上，完成数据的绑定，以及后续数据的操作等。
- ❑ **输入：**由用户完成的输入事件，然后通过控件检测之后发出响应。WPF 输入系统使用直接事件和路由事件来支持文本输入、焦点管理和鼠标定位。
- ❑ **命令：**应用程序通常具有复杂的输入要求。WPF 提供了一个命令系统，它将用户输入操作与响应这些操作的代码相分离。
- ❑ **图形：**WPF 在处理图形功能上，可谓是其强项所在，它引进了一组广泛的、灵活的、可扩展的图形处理功能。具有如下优点，与分辨率和设备无关的图形；更高的精度；高级图形和动画支持；硬件加速。除此之外，它还在处理二维形状、二维几何图形、三维呈现等图形展现上有强大的支持。
- ❑ **动画：**WPF 动画支持可以使控件变大、旋转、调节和淡化，以产生有趣的页面过渡和更多效果。可以对大多数 WPF 类或者是自定义类进行动画处理。
- ❑ **媒体：**媒体的应用包括图像、视频和音频。WPF 对于这些媒体的应用提供了特殊的支持。
- ❑ **文档：**WPF 支持使用三种类型的文档，即流文档、固定文档和 XML 纸张规范（XPS）文档。WPF 还提供了用于创建、查看、管理、批注、打包和打印文档的服务。
- ❑ **自定义应用程序：**WPF 提供了强大的自定义功能，通过该功能，开发人员可以创建出符合自身需求的通用的模板，以及个性化特点的应用程序。可自定义部分包括内容模型、控件模板、数据模板、样式、资源、主题和资源、自定义控件等。

27.2 创建 WPF 应用程序

同.NET 其他的应用程序相同, WPF 也包括几种不同的 WPF 应用程序类型, 这些应用程序可以通过 Visual Studio 2010 进行创建和开发。本节将介绍 WPF 应用程序的相关编程, 以及程序的创建过程。

27.2.1 创建 WPF 的过程

WPF 程序开发可以通过 Visual Studio 2010 中的项目创建来实现, Visual Studio 2010 可以创建 4 种 WPF 类型的项目, 分别是 WPF 应用程序、WPF 浏览器应用程序、WPF 用户控件库和 WPF 自定义控件库。

注意: WPF 应用程序类似客户端服务器模型的应用程序, 需要单独的客户端工具。而 WPF 浏览器应用程序是可以运行在浏览器的应用程序。

下面通过一个创建 WPF 应用的过程, 来说明 WPF 项目的项目结构, 以及 Visual Studio 2010 中 WPF 的开发界面组成。

(1) 创建 WPF 应用程序。启动 Visual Studio 2010 开发工具。启动之后如图 27.2 所示。



图 27.2 Visual Studio 2010 启动窗口


技巧: 启动窗口可以通过菜单栏的“工具”|“选项”菜单中的“选项”对话框中的“环境”|“启动”命令对话框进行设置。

选择菜单栏中的“文件”|“新建”|“项目”命令，弹出“新建项目”的对话框，如图 27.3 所示。



图 27.3 新建项目


在“新建项目”对话框中，左侧的“已安装模板”区域列出了 Visual Studio 2010 可以创建的所有类型，右侧的“模板”区域列出了对应该项目类型所包含的模板信息。如果需要创建 WPF 应用程序，在“已安装模板”区域选择 Windows，在右侧“模板”区域选择“WPF 应用程序”。在“名称”栏输入该项目的名称，“位置”栏输入该项目保存的路径，然后单击“确定”按钮。

 **说明：**若要创建 WPF 浏览器应用程序，需要在“模板”区域选择“WPF 浏览器应用程序”选项。

(2) 查看项目结构组成。项目创建完毕后，可以在 Visual Studio 2010 中，通过“解决方案资源管理器”查看该项目的文件结构，如图 27.4 所示。



图 27.4 解决方案资源管理器

 **注意：**如果“解决方案资源管理器”窗口默认没有显示出来，可以选择菜单栏中的“视图”|“解决方案资源管理器”命令，打开“解决方案资源管理器”窗口。

(3) 程序开发。单击图 27.4 中的 MainWindow.xaml 文件节点，弹出设计窗口和 XAML 窗口，如图 27.5 所示。

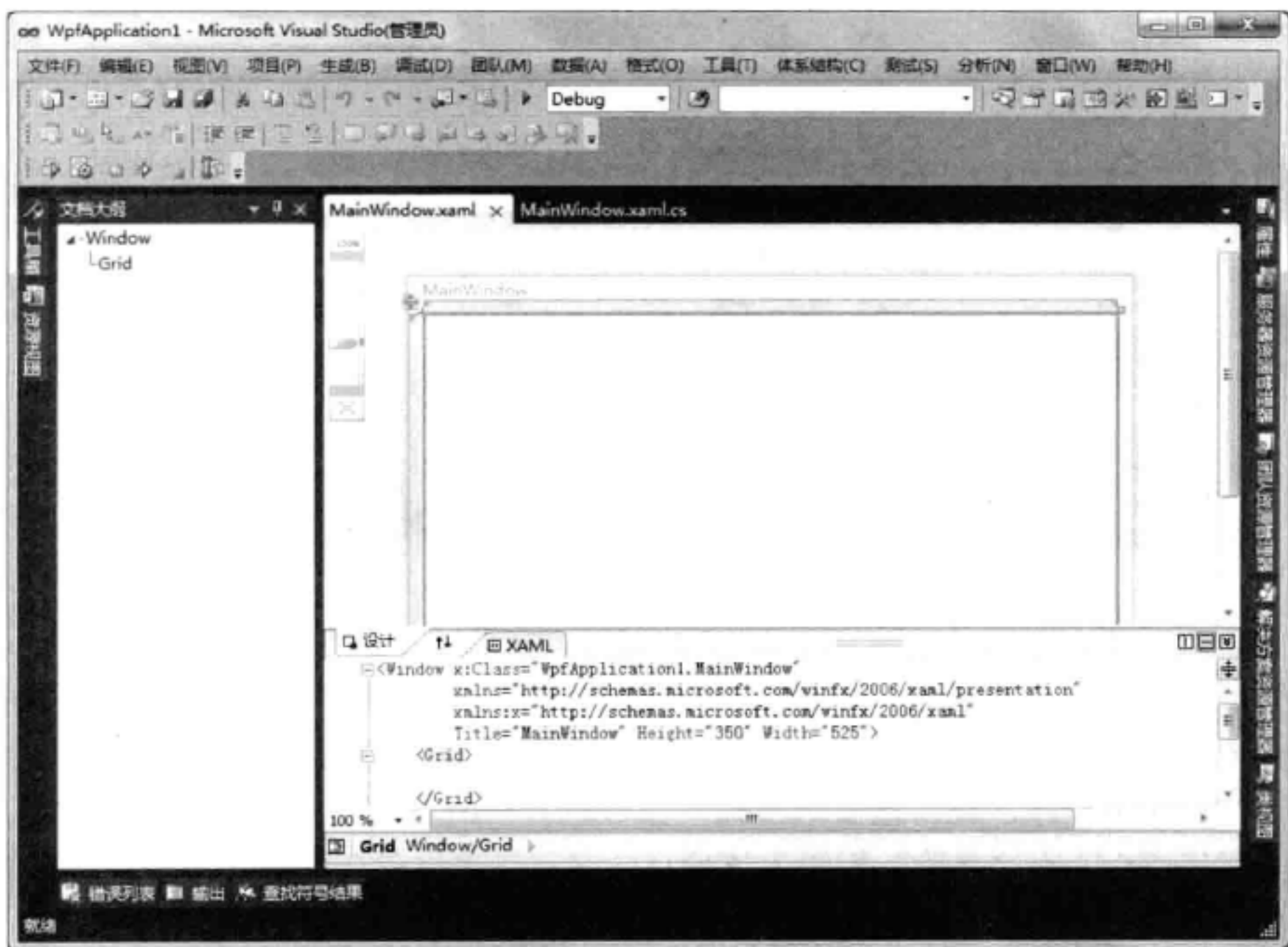



图 27.5 “设计”视图和 XAML 视图


“设计”视图，用于可视化的界面设计，开发人员可以直接通过拖曳“工具箱”中的控件到“设计”视图中，完成控件的添加、XAML 视图用来显示对应设计视图界面的 XAML 代码，也可以通过 XAML 视图的可视化编辑，来完成控件的添加。

 **注意：**“设计”视图与 XAML 视图是同步的，即在“设计”视图添加一个控件，XAML 视图将自动添加该控件对应的 XAML 代码。

如果在“设计”视图里添加一个按钮控件，添加之后的“设计”视图和 XAML 视图如图 27.6 所示。

单击图 27.4 中的 MainWindow1.xaml.cs 文件节点，弹出 MainWindow1 对应的后台代码编写窗口，如图 27.7 所示。

通过图 27.7 的代码编辑窗口，可以编写该文件对应的后台操作代码。

 **技巧：**除了通过“解决方案资源管理器”窗口双击该 cs 文件打开代码编辑窗口之外，还可以在“设计”视图里，右键单击“查看代码”菜单，打开代码编辑窗口。

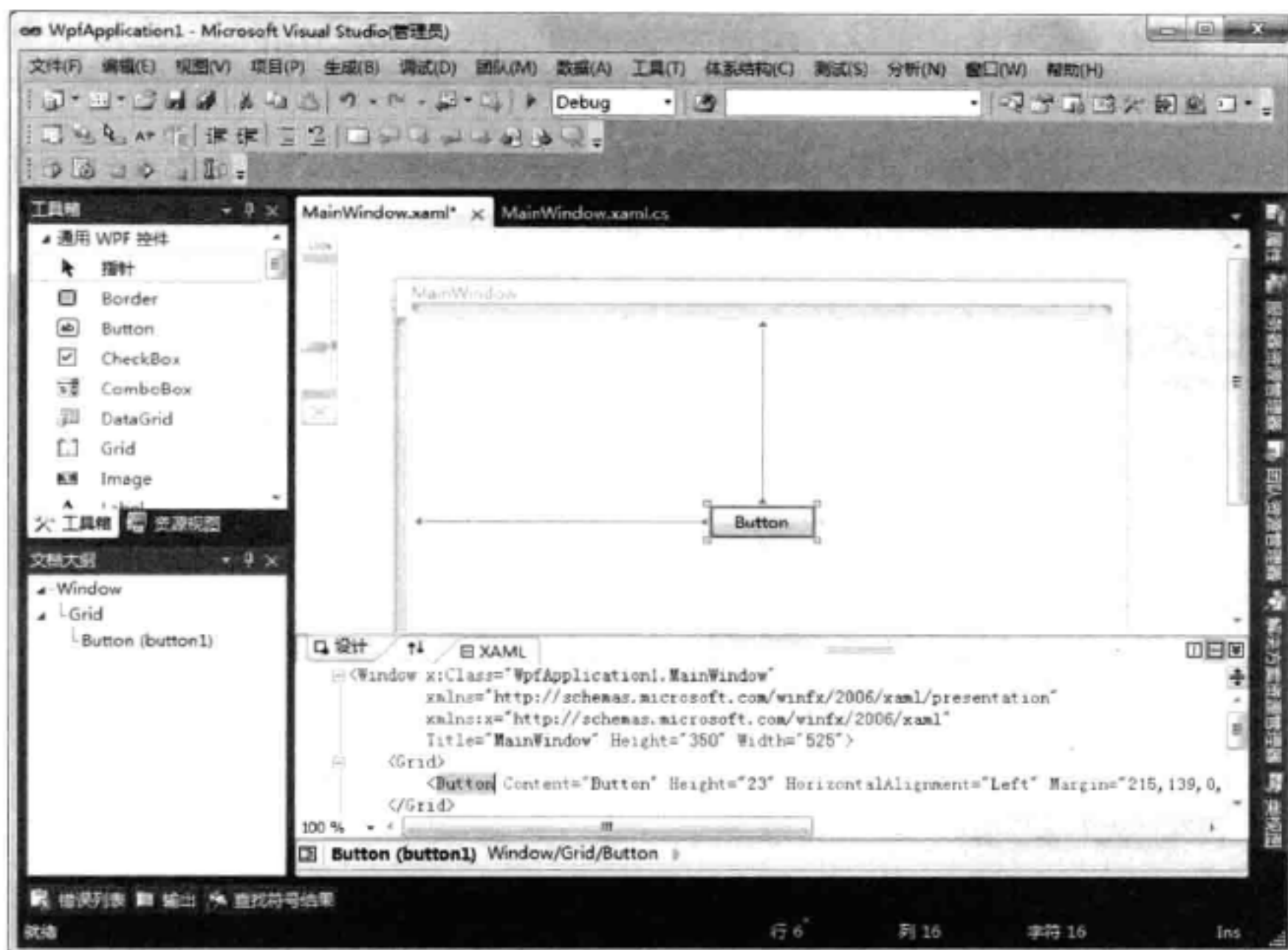


图 27.6 添加控件之后的“设计”视图和 XAML 视图

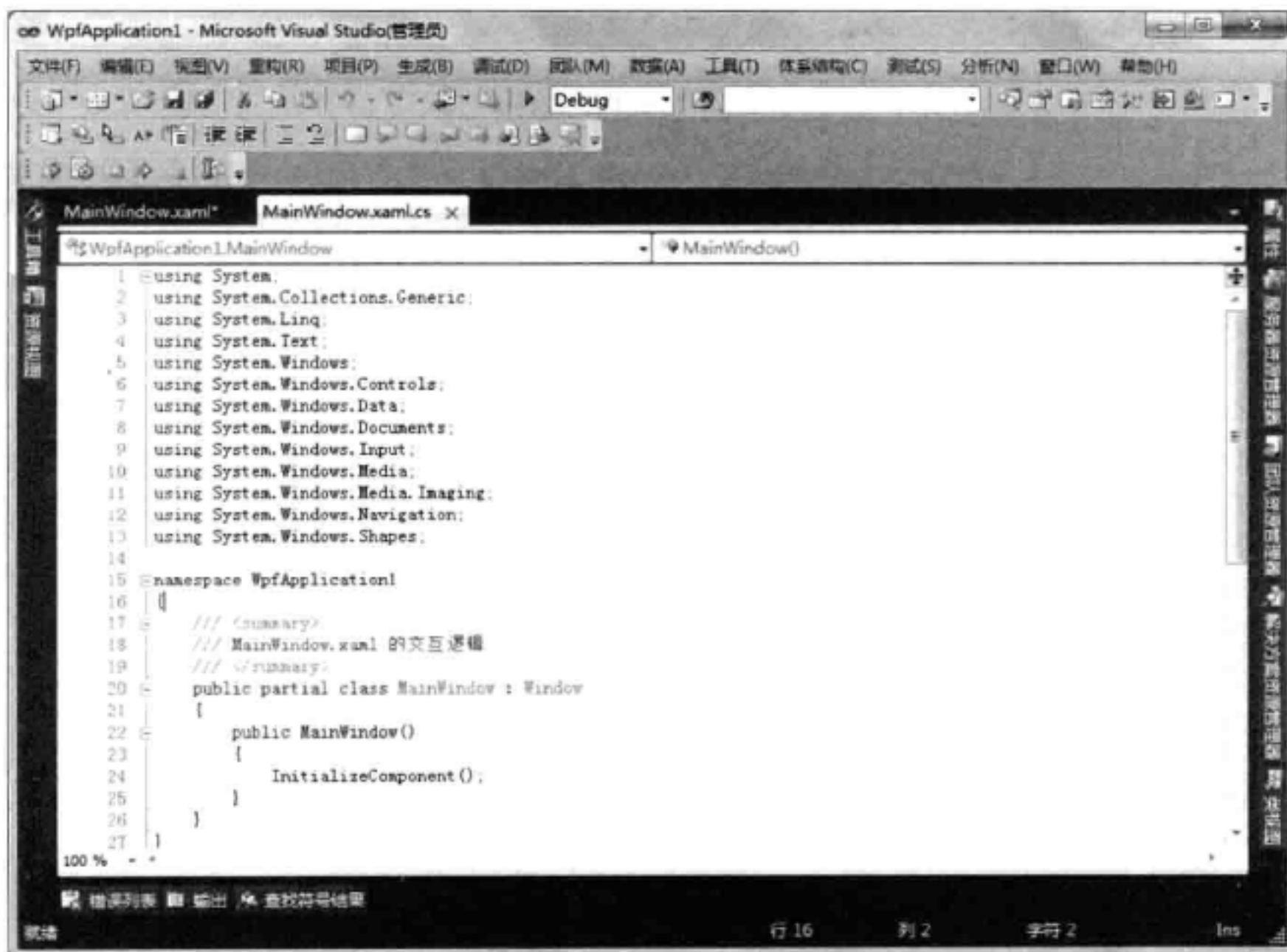


图 27.7 代码编辑窗口

27.2.2 完整的 WPF 应用程序实例

WPF 应用程序就是通常所说的 Windows 应用程序，是用于构件客户端/服务器应用模

型的应用程序类型。本节将以实例介绍如何通过 Visual Studio 2010 创建完整的 WPF 应用程序。

(1) 创建 WPF 应用程序项目。参照 27.2.1 节中的步骤 (1)，创建 WPF 应用程序，将“名称”设置为 HelloWorldWindowsApp，如图 27.8 所示。



图 27.8 创建 WPF 应用程序

(2) 添加控件。在“解决方案资源管理器”窗口中，双击 MainWindow1.xaml 文件，在 XAML 窗口输入如下代码：

```
<!--定义 Window 窗体-->
<Window x:Class="HelloWorldWindowsApp.MainWindow"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    Title="HelloWorld" Height="300" Width="300">
    <!--定义 Window 窗体的标题,高度和宽度-->
    <!--定义网格控件-->
    <Grid>
        <!--定义按钮控件-->
        <Button Name="button" Width="100" Height="20" Click="button_Click">
            点击!</Button>
        </Grid>
    </Window>
```

此段代码表示在窗体中添加了一个 Grid 网格控件，在网格中添加了一个按钮控件，分别定义了该按钮控件的名称、宽度、高度、显示名称等属性，同时定义了该按钮的一个单击事件，名称为 button_Click。

(3) 添加事件处理代码。双击 MainWindow.xaml.cs 文件，打开该文件的后台代码编辑窗口。在 MainWindow 类中添加如下代码：


```
//定义分部类
public partial class MmainWindow : Window
{
    //构造函数
    public MainWindow()
    {
        //初始化窗体组件
        InitializeComponent();
    }

    //定义按钮单击事件
    private void button_Click(object sender, RoutedEventArgs e)
    {
        //弹出提示窗口
        MessageBox.Show("Hello World !");
    }
}
```

方法 `button_Click` 为事件响应代码，表示该按钮被单击之后的代码处理逻辑，方法内部表示单击该按钮之后，弹出一个消息框，显示“Hello World!”文本。

(4) 运行该项目。单击菜单栏“调试”|“启动调试”菜单，Visual Studio 2010 开始编译该项目，同时弹出运行结果。单击“点击”按钮之后，运行结果如图 27.9 所示。



图 27.9 WPF 应用程序运行效果

技巧：除了通过菜单栏的功能菜单完成程序运行之外，还可以通过按 F5 键实现运行。

27.2.3 创建 WPF 浏览器应用程序

与 WPF 应用程序不同，WPF 浏览器应用程序就是通常所说的基于浏览器的应用程序，其是用于构建浏览器端/服务器应用模型的应用程序类型。本节将以实例介绍如何通过 Visual Studio 2010 创建完整的 WPF 浏览器应用程序。

(1) 创建 WPF 浏览器应用程序项目。参照 27.2.1 节中的步骤 (1)，创建 WPF 浏览器应用程序，将“名称”设置为 `HelloWorldPagesApp`，如图 27.10 所示。



图 27.10 创建 WPF 浏览器应用程序

(2) 添加控件。在“解决方案资源管理器”窗口中，双击 Page1.xaml 文件，在 XAML 窗口输入如下代码：

```
<!--定义 Page 页面-->
<Page x:Class="HelloWorldPagesApp.Page1"<!--定义窗体的类名称-->
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    <!--定义 xmlns 命名空间-->
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    <!--定义 xmlns:x 命名空间-->
    Title="Page1"><!--定义窗体的标题 -->
    <!--定义网格控件-->
    <Grid>
        <!--定义按钮控件-->
        <Button Name="button" Width="100" Height="20" Click="button_Click">
            点击!</Button>
        </Grid>
    </Page>
```

可以参考 WPF 应用程序中的代码，对比发现，二者除了根元素有区别之外，其他内容全部一样。这与以前的程序开发区别很大，说明 WPF 在处理基本应用程序和浏览器应用程序的代码部分可以重复利用。

(3) 添加事件处理代码。双击 Page1.xaml.cs 文件，打开该文件的后台代码编辑窗口。在 Page1 类中添加如下代码：

```
//定义分部类
public partial class Page1 : Page
{
    //构造函数
    public Page1()
    {
        //初始化窗体组件
        InitializeComponent();
    }
}
```



```
}  
//定义按钮单击事件  
private void button_Click(object sender, RoutedEventArgs e)  
{  
    //弹出提示窗口  
    MessageBox.Show("Hello World !");  
}  
}
```

方法 `button_Click` 为事件响应代码，表示该按钮被单击之后的代码处理逻辑，方法内部代码表示单击该按钮之后弹出一个消息框，显示“Hello World!”文本。该类与 WPF 应用程序比较，区别在于实现窗口的类继承不同，WPF 应用程序继承于 `Window` 类，而该浏览器应用程序的实现类继承于 `Page` 类。

(4) 运行该项目。单击菜单栏“调试”|“启动调试”菜单，Visual Studio 2010 开始编译该项目，同时弹出运行结果。单击“点击”按钮之后，运行结果如图 27.11 所示。



图 27.11 WPF 浏览器应用程序运行效果

注意：WPF 浏览器应用程序与 WPF 应用程序不同，它是以浏览器作为运行载体的。

27.3 简单 WPF 实例

WPF 是一个很广泛的框架，其中涉及的技术很多，本章将从几个重要的方面介绍 WPF 的技术组成与具体实现。

27.3.1 用 `ListBox` 控件实现列表显示

WPF 是实现系统页面展现的框架，所以 WPF 的展现系统中，控件起着重要的作用，可以说众多绚丽的 WPF 页面效果都是由基本的 WPF 控件结合其他技术实现的。在前面的

WPF 的应用程序中,已经介绍了按钮控件,以及相关事件,本节将通过一个完整的实例,介绍 ListBox 选择控件,以及简单介绍实现控件的 XAML 语言。

(1) 创建项目名称为 ListBoxSample 的 WPF 应用程序。

(2) 在项目中的 MainWindow.xaml 文件的 XAML 编辑视图中,首先添加 Window 窗体的代码定义。示例代码如下:

```
<!--定义 Window 窗体-->
<Window x:Class="ListBoxSample.MainWindow"
    <!--定义窗体的类名称-->
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    <!--定义 xmlns 命名空间-->
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    <!--定义 xmlns:x 命名空间-->
    Title="ListBox 测试" Height="286" Width="294">
    <!--定义窗体的标题,高度和宽度-->
```

其中的 Window 节点表示该页面属于 WPF 应用程序的窗口定义文件, X:Class 属性表示该 XAML 文件对应的后台类的名称, 此实例的后台代码文件的命名空间为 ListBoxSample, 类名称为 MainWindow。Xmlns 属性表示该 XAML 文件定义的命名空间, 默认都是标准的 http://schemas.microsoft.com/winfx/2006/xaml/presentation, Title 属性表示该窗口的名称, Height 和 Width 属性分别表示该窗口的高度和宽度。

(3) 添加布局控件。接下来在 XAML 视图中再添加一个 Canvas 布局控件。布局控件是用来对页面的控件位置进行控制的位置控件。代码如下:

```
<!--定义布局控件-->
<Canvas>
    <!--定义布局控件资源-->
    <Canvas.Resources>
        <!--定义 ListBox 控件属性-->
        <Style x:Key="Simple" TargetType="{x:Type ListBox}">
            <!--定义 SelectionMode 属性和值-->
            <Setter Property="SelectionMode" Value="Single"/>
            <!--定义 Background 属性和值-->
            <Setter Property="Background" Value="Red"/>
        </Style>
        <!--定义 ListBoxItem 控件属性-->
        <Style x:Key="SimpleListBoxItem"
            TargetType="{x:Type ListBoxItem}">
            <!--定义 FontSize 属性和值-->
            <Setter Property="FontSize" Value="14"/>
            <!--定义 Background 属性和值-->
            <Setter Property="Background" Value="Pink"/>
            <!--定义 Foreground 属性和值-->
            <Setter Property="Foreground" Value="Purple"/>
        </Style>
        <!--定义 ListBoxItem 触发属性-->
        <Style x:Key="Triggers" TargetType="{x:Type ListBoxItem}">
            <Style.Triggers>
                <!--定义 IsMouseOver 事件属性-->
                <Trigger Property="ListBoxItem.IsMouseOver"
                    Value="true">
                    <!--定义 Foreground 属性和值-->
                    <Setter Property="Foreground" Value="Red"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Canvas.Resources>

```



```

        <!--定义 Background 属性和值-->
        <Setter Property = "Background" Value="LightBlue"/>
    </Trigger>
</Style.Triggers>
</Style>
</Canvas.Resources>

```

在该 Canvas 布局控件中，定义了资源文件，资源文件的语法为 Canvas.Resources，然后通过样式 Style 节点进行描述。3 个样式对应的键值名称分别为 Simple、SimpleListBoxItem 和 Triggers。但 3 个样式对应的控制控件类型 TargetType 却不一样，Simple 对应控制的是 ListBox 控件的样式，而 SimpleListBoxItem 和 Triggers 控制的是 ListBoxItem 的样式，关键字 x:Type 表示针对的对象类型，x 则表示前面已经定义过的 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" 命名空间。在 Style 中定义该指定控件的具体属性，比如 Simple 样式中，定义了 ListBox 控件的 SelectionMode 属性为 Single，即只能单选，同时定义了 Background 属性为 Red，即背景色为红色。除了可以定义样式的静态属性之外，还可以定义样式的触发器属性，如 Triggers 属性的<Style.Triggers>节点，定义了触发器的激活动作为 IsMouseOver，即在鼠标经过 ListBoxItem 子项的时候，样式被激活，激活的效果是前景色为 Red，背景色为 LightBlue。

(4) 添加网格控件。接下来，在页面添加一个 Grid 网格控件，用于放置 3 个 ListBox 控件。代码如下：

```

<!--定义网格控件-->
<Grid Name="grid" ShowGridLines ="false" Background ="White" >
    <!--定义网格列-->
    <Grid.ColumnDefinitions>
        <!--定义网格列的宽度为 300-->
        <ColumnDefinition Width="300"/>
    </Grid.ColumnDefinitions>
    <!--定义网格行-->
    <Grid.RowDefinitions>
        <!--定义网格为 5 行-->
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions><!--行定义控件结束-->
    <!--添加 DockPanel 控件位置为 0 列 2 行-->
    <DockPanel Margin="10, 10, 3, 3" Grid.Column="0" Grid.Row="2">
        <!--添加 ListBox 控件,使用静态资源 Simple -->
        <ListBox Style="{StaticResource Simple}" Width="100"
            Height="55" >
            <!--添加 ListBoxItem 子项-->
            <ListBoxItem>选项 1</ListBoxItem>
            <ListBoxItem>选项 2</ListBoxItem>
            <ListBoxItem>选项 3</ListBoxItem>
            <ListBoxItem>选项 4</ListBoxItem>
            <ListBoxItem>选项 5</ListBoxItem>
            <ListBoxItem>选项 6</ListBoxItem>
            <ListBoxItem>选项 7</ListBoxItem>
            <ListBoxItem>选项 8</ListBoxItem>
            <ListBoxItem>选项 9</ListBoxItem>
        </ListBox>
    </DockPanel>
</Grid>

```



```

        <ListBoxItem>选项 10</ListBoxItem>
    </ListBox><!-- ListBox 控件结束-->
</DockPanel><!-- DockPanel 控件结束-->
<!--添加 DockPanel 控件位置为 0 列 3 行-->
<DockPanel Margin="10, 10, 3, 3" Grid.Column="0" Grid.Row="3">
    <!--添加 ListBox 控件-->
    <ListBox Width="260" Height="55" HorizontalAlignment="Left">
        <!--添加 ListBoxItem 子项, 使用静态资源 SimpleListBoxItem -->
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 1</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 2</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 3</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 4</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 5</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 6</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 7</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 8</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 9</ListBoxItem>
        <ListBoxItem Style="{StaticResource SimpleListBoxItem}">
            选项 10</ListBoxItem>
    </ListBox>
</DockPanel>
<!--添加 DockPanel 控件位置为 0 列 4 行-->
<DockPanel Margin="10, 10, 3, 3" Grid.Column="0" Grid.Row="4">
    <!--添加 ListBox 控件, 定义了宽度和高度以及左对齐-->
    <ListBox Width="260" Height="55" HorizontalAlignment="Left">
        <!--添加 ListBoxItem 子项, 使用静态资源 Triggers -->
        <ListBoxItem Style="{StaticResource Triggers}">选项 1
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 2
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 3
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 4
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 5
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 6
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 7
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 8
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 9
        </ListBoxItem>
        <ListBoxItem Style="{StaticResource Triggers}">选项 10
        </ListBoxItem>
    </ListBox>
</DockPanel>

```



```

</Grid>
</Canvas>
</Window>

```

该 Grid 控件为 5 行 1 列，然后将 DockPanel 布局控件添加到 Grid 控件中已经划分好的行单元格中，Grid.Column 属性和 Grid.Row 属性指定了该 DockPanel 控件所在 Grid 控件中的位置。添加完 3 个 DockPanel 之后，再将 3 个 ListBox 控件放置到 DockPanel 之中，然后通过 Style="{StaticResource **}" 语法对各个控件的样式进行指定。

(5) 运行该实例的效果如图 27.12 所示。



图 27.12 ListBox 控件使用效果

27.3.2 用 Hyperlink 控件实现多页面切换

Hyperlink 控件是用来实现超级链接功能的控件，使用该控件可以完成多个内部页面之间的导航或者直接通过浏览器打开外部网站的页面。

说明：Hyperlink 控件可以实现类似于网页的链接效果，但不同的是，该控件也可以在客户/服务端应用程序模型中使用。

下面通过一个实例介绍如何使用 Hyperlink 控件实现多个页面切换。

- (1) 创建 WPF 应用程序，项目名称为 MultiPageSample。
- (2) 打开 Page1.xaml 文件的 XAML 视图编辑，在页面中添加如下代码：

```

<!-- 定义 StackPanel 控件,背景色为 LightBlue -->
<StackPanel Background="LightBlue">
    <!--TextBlock 控件,定义 Margin 属性-->
    <TextBlock Margin="10,10,10,10">起始页</TextBlock>
    <!--TextBlock 控件,定义 Margin 属性以及左对齐-->
    <TextBlock HorizontalAlignment="Left" Margin="10,10,10,10">
        <!--添加超级链接控件,链接目标文件为 Page2.xaml -->
        <Hyperlink NavigateUri="Page2.xaml">下一页</Hyperlink>
    </TextBlock>
</StackPanel>

```

在页面中添加了一个 Hyperlink 控件，该控件的 NavigateUri 属性设置为链接的目标页面名称，同时指定了链接的文本为“下一页”。

- (3) 添加一个 Page 页面，命名为 Page2.xaml，同样在 XAML 视图编辑中添加如下代码：

```
<StackPanel Background="LightGreen">
    <!--TextBlock 控件,定义位置为上停靠同时定义 Margin 属性-->
    <TextBlock DockPanel.Dock="Top"
        Margin="10,10,10,10">第二页</TextBlock>
    <!--TextBlock 控件,定义位置为左对齐,同时定义 Margin 属性-->
    <TextBlock HorizontalAlignment="Left" Margin="10,10,10,10">
        <!--添加超级链接控件,链接目标文件为 Page1.xaml -->
        <Hyperlink NavigateUri="Page1.xaml">跳转到起始页</Hyperlink>
    </TextBlock>
</StackPanel>
```

该页面中也设置了一个 Hyperlink 控件，该控件的链接目标页面为 Page1.xaml，文本为“跳转到起始页”，这样便实现了两个页面之间的链接切换。

(4) 运行该实例，效果如图 27.13 和图 27.14 所示。

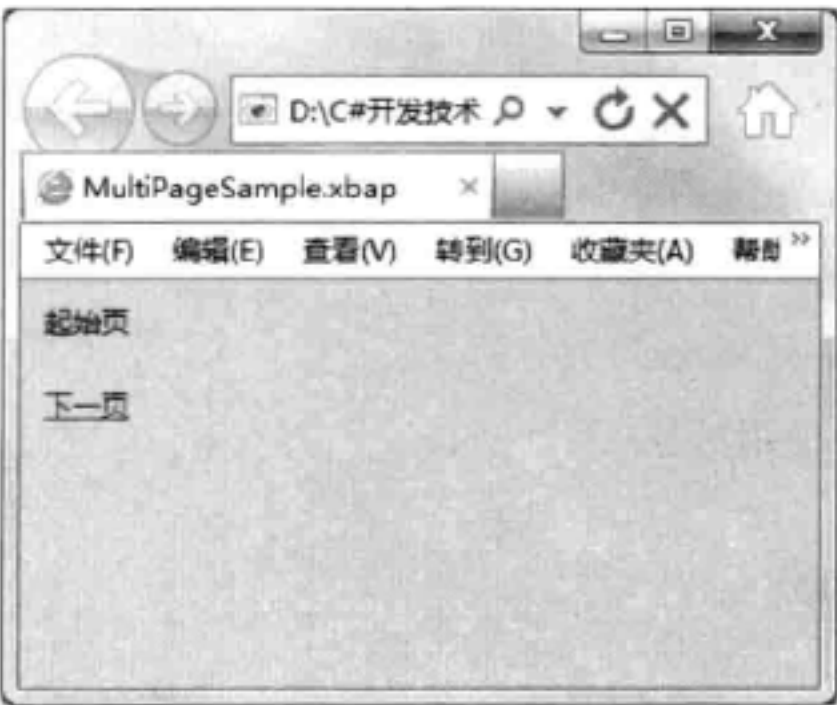


图 27.13 Hyperlink 控件的首页

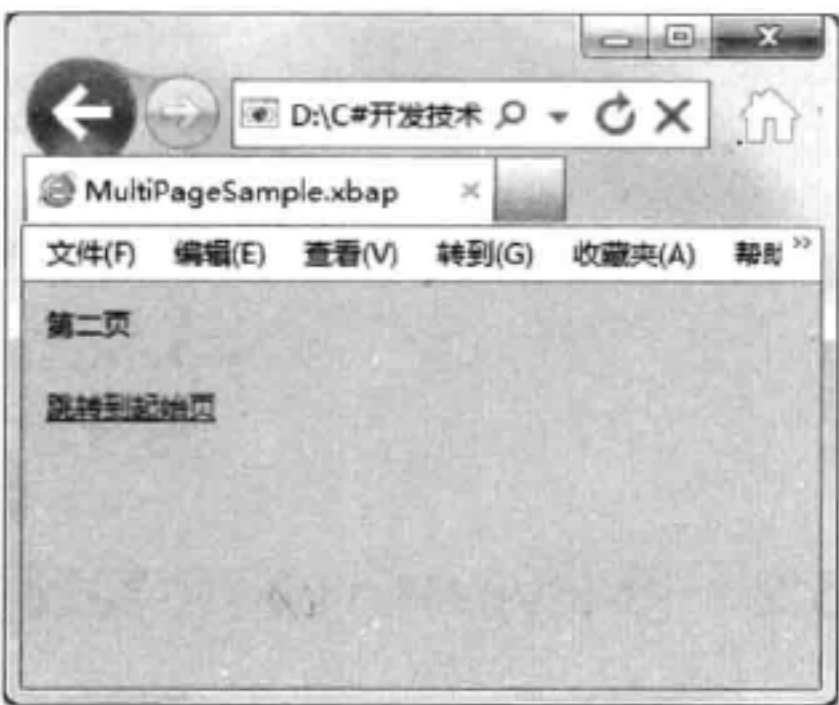


图 27.14 Hyperlink 控件的页面切换

27.3.3 用 DockPanel 沿容器边缘定位

布局控件是用来控制页面控件的摆放位置。布局控件有很多种，具体各个控件的名称与说明如表 27.1 所示。

表 27.1 布局控件说明

名 称	说 明
Canvas	定义一个区域，在此区域内，可以使用相对于Canvas区域的坐标
DockPanel	定义一个区域，在此区域中，可以使子元素互相水平或垂直排列
Grid	定义由行和列组成的灵活网格区域
StackPanel	将子元素排列成一行
VirtualizingPanel	为其子数据集合的Panel元素提供一个框架
WrapPanel	从左至右按顺序位置定位子元素，在包含框的边缘处将内容断开至下一行

本节和 27.3.4 节将分别用实例说明两种布局控件，DockPanel 控件和 StackPanel 控件。DockPanel 控件用于沿布局容器的边缘定位子内容。开发用户界面时，如果想将某个控件停靠到窗口一侧，例如，菜单栏经常停靠在窗口的顶部或一侧；可以使用 DockPanel 容器将控件停靠到窗口侧面。

实例的创建过程如下所述。

(1) 创建 WPF 浏览器应用程序，项目名称为 LayoutSample。

(2) 打开 Page1.xaml 文件的 XAML 视图编辑, 在页面中添加如下代码:

```
<!--定义 DockPanel 控件-->
<DockPanel >
    <!--添加 TextBlock 控件, 背景色为 LightBlue, 上停靠, 内部文本为"标题栏"-->
    <TextBlock Background="LightBlue"
        DockPanel.Dock="Top">标题栏</TextBlock>
    <!--添加 TextBlock 控件, 背景色为 LightYellow, 下停靠, 内部文本为 "状态栏"-->
    <TextBlock DockPanel.Dock="Bottom"
        Background="LightYellow">状态栏.</TextBlock>
    <!--添加 TextBlock 控件, 背景色为 Lavender, 左停靠, 内部文本为"菜单栏"-->
    <TextBlock DockPanel.Dock="Left"
        Background="Lavender">菜单栏</TextBlock>
    <!--添加子 DockPanel 控件, 背景色为 Bisque -->
    <DockPanel Background="Bisque">
        <!--添加 StackPanel 控件, 上停靠-->
        <StackPanel DockPanel.Dock="Top">
            <!--添加 Button 控件, 定义左对齐, 高度为 30, 宽度为 100, 定义 Margin 属性, 内部文本为"登录"-->
            <Button HorizontalAlignment="Left"
                Height="30px"
                Width="100px"
                Margin="10,10,10,10">登录</Button>
            <!--添加 Button 控件, 定义左对齐, 高度为 30, 宽度为 100, 定义 Margin 属性, 内部文本为"注册"-->
            <Button HorizontalAlignment="Left"
                Height="30px"
                Width="100px"
                Margin="10,10,10,10">注册</Button>
        </StackPanel><!-- StackPanel 控件结束-->
        <!--添加 TextBlock 控件, 背景色为 LightGreen, 内部文本-->
        <TextBlock Background="LightGreen">该布局采用的是 DockPanel 控件</TextBlock>
    </DockPanel><!--子 DockPanel 控件结束-->
</DockPanel><!-- DockPanel 控件结束-->
```

从以上代码可以看出, DockPanel 将该页面分为 4 个部分, 分别是 DockPanel.Dock="Top"的标题栏、DockPanel.Dock="Left"的菜单栏、子区域的内容栏和 DockPanel.Dock="Bottom"的状态栏。4 个部分分别用不同颜色加以区别。

(3) 运行该实例, 效果如图 27.15 所示。



图 27.15 DockPanel 控件使用效果

27.3.4 使用 StackPanel 叠放包含的控件

StackPanel 控件是另外一种常用的布局控件。StackPanel 要么垂直叠放包含的控件，要么将包含的控件排列在水平行中，具体情况取决于 Orientation 属性的值。如果将比 StackPanel 的宽度能显示的控件还要多的控件添加到 StackPanel 中，这些控件将被截掉且不显示。

 **注意：**StackPanel 控件的 Orientation 属性默认为垂直放置。

下面通过一个实例介绍如何使用 StackPanel 控件进行布局。

(1) 创建 WPF 应用程序，项目名称为 StackPanelSample。

(2) 打开 MainWindow.xaml 文件的 XAML 视图编辑，在页面中添加如下代码：

```
<StackPanel>
    <!--添加#1 区域,定义背景色,边界画刷,边界厚度-->
    <Border Background="SkyBlue" BorderBrush="Black"
        BorderThickness="1">
        <!--添加 TextBlock 控件,定义前景色,字体大小以及内部文本-->
        <TextBlock Foreground="black" FontSize="12">#1</TextBlock>
    </Border>
    <!--添加#2 区域,定义背景色,边界画刷,边界厚度-->
    <Border Width="400" Background="CadetBlue" BorderBrush="Black"
        BorderThickness="1">
        <!--添加 TextBlock 控件,定义前景色,字体大小以及内部文本-->
        <TextBlock Foreground="black" FontSize="14">#2</TextBlock>
    </Border>
    <!--添加#3 区域,定义背景色,边界画刷,边界厚度-->
    <Border Background="#ffff99" BorderBrush="Black"
        BorderThickness="1">
        <!--添加 TextBlock 控件,定义前景色,字体大小以及内部文本-->
        <TextBlock Foreground="black" FontSize="16">#3</TextBlock>
    </Border>
    <!--添加#4 区域,定义背景色,边界画刷,边界厚度-->
    <Border Width="200" Background="PaleGreen" BorderBrush="Black"
        BorderThickness="1">
        <!--添加 TextBlock 控件,定义前景色,字体大小以及内部文本-->
        <TextBlock Foreground="black" FontSize="18">#4</TextBlock>
    </Border>
    <!--添加#5 区域,定义背景色,边界画刷,边界厚度-->
    <Border Background="White" BorderBrush="Black"
        BorderThickness="1">
        <!--添加 TextBlock 控件,定义前景色,字体大小以及内部文本-->
        <TextBlock Foreground="black" FontSize="20">#5</TextBlock>
    </Border>
</StackPanel>
```

从以上代码可以看出，在 StackPanel 控件中，一共放置了 5 个 TextBlock 控件，这些控件设置了各自的属性之后，全部垂直排列在页面中。

(3) 运行该实例，效果如图 27.16 所示。



图 27.16 StackPanel 控件使用效果

27.3.5 使用数据源集合实现多数据绑定

数据绑定是指通过数据绑定控件实现控件与数据源之间的数据同步，这里所说的数据源有很多种类型，例如数据库、对象、文件或者一个 XML 数据等。本节通过一个实例介绍如何对控件实现数据的绑定，同时完成将多个数据源的数据统一绑定到数据控件中。

- (1) 创建 WPF 应用程序，项目名称为 MultiDataBinding。
- (2) 打开 MainWindow.xaml 文件的 XAML 视图编辑，在页面中添加如下代码：

```
<!--定义 Window 窗体-->
<Window Background="Cornsilk"
    <!--定义 Window 窗体的背景色-->
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    <!--定义 xmlns 命名空间-->
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    <!--定义 xmlns:x 命名空间-->
    xmlns:c="clr-namespace:MultiDataBinding"
    <!--定义 xmlns:c 命名空间-->
    x:Class="MultiDataBinding.MainWindow"<!--定义窗体的类名称-->
    Title="多数据绑定实例"<!--定义窗体的标题,高度和宽度-->
    SizeToContent="WidthAndHeight"<!--定义窗体的布局方式-->
>
```

该段 XAML 代码描述了 Window 窗体的基本属性，同前面几个实例不同，本节中定义了 xmlns:c 的属性。该属性值指定了后续 XAML 代码中，以 c:为前缀的代码部分的类所在的命名空间。

- (3) 添加资源代码。接下来添加一段关于数据源的资源代码，代码如下所示。

```
<!--窗口资源定义-->
<Window.Resources>
    <!--添加对象数据源-->
    <c:Source1 x:Key="Source1Data"/>
    <!--添加 XML 数据源 XmlDataProvider 对象-->
    <XmlDataProvider x:Key="Source2Data" XPath="Source2/Source2Item">
        <!--添加数据类定义-->
        <x:XData>
            <!-- 定义 Source2 命名空间属性 -->
            <Source2 xmlns="">
```

```

        <!--定义数据内容名称属性 -->
        <Source2Item Name="Source1 选项 1" />
        <Source2Item Name="Source2 选项 2" />
        <Source2Item Name="Source2 选项 3" />
        <Source2Item Name="Source2 选项 4" />
        <Source2Item Name="Source2 选项 5" />
    </Source2>
</x:XData>
</XmlDataProvider><!-- XmlDataProvider 对象结束-->
<!--添加数据模板, 数据类型为 Source1Item -->
<DataTemplate DataType="{x:Type c:Source1Item}">
    <!--添加 TextBlock 控件, 内部文本绑定属性为 Name, 前景色为 Gold-->
    <TextBlock Text="{Binding Path=Name}" Foreground="Gold"/>
</DataTemplate><!--数据模板定义结束 -->
<!--添加数据模板, 数据类型为 Source2Item -->
<DataTemplate DataType="Source2Item">
    <!--添加 TextBlock 控件, 内部文本绑定属性为 Name, 前景色为 Cyan -->
    <TextBlock Text="{Binding XPath=@Name}" Foreground="Cyan"/>
</DataTemplate><!--数据模板定义结束 -->
</Window.Resources><!--窗口资源定义结束-->

```

代码中定义了两个数据源, 它们的键值分别为 Source1Data 和 Source2Data, 其中 Source1Data 数据源为对象数据源, 它对应了后台代码中的 Source1 类实例。Source2Data 数据源为 XML 数据源, 它直接通过 XData 属性定义了一个以 Source2 为根节点的 XML 数据源。

(4) 添加数据模板定义。接下来通过 DataTemplate 定义了绑定数据控件的数据模板, 指明了绑定对象的类型, 以及绑定文本的内容和文本的前景色属性。定义完数据绑定模板之后, 需要添加页面的显示控件代码, 添加的代码如下:

```

<!--定义 StackPanel 控件-->
<StackPanel>
<!--添加 TextBlock 控件, 字大小为 18, 字重量为 Bold, 边距 10, 居中对齐, 内部文本为" 多数据绑定"-->
<TextBlock FontSize="18" FontWeight="Bold" Margin="10"
    HorizontalAlignment="Center">多数据绑定</TextBlock>
    <!--添加 ListBox 控件-->
    <ListBox Name="myListBox" Height="300" Width="200"
        Background="White">
        <!--添加 ListBox 数据源-->
        <ListBox.ItemsSource>
            <!--启用要以单个列表形式显示的多个集合和项-->
            <CompositeCollection>
                <!--绑定数据源 Source1Data 数据内容 -->
                <CollectionContainer Collection="{Binding Source=
                    {StaticResource Source1Data}}"/>
                <!--绑定数据源 Source2Data 数据内容 -->
                <CollectionContainer Collection="{Binding Source=
                    {StaticResource Source2Data}}"/>
                <!--绑定 ListBoxItem 子项, 前景色为 Red, 文本为 "其他绑定 1"-->
                <ListBoxItem Foreground="Red">其他绑定 1</ListBoxItem>
                <!--绑定 ListBoxItem 子项, 前景色为 Red, 文本为 "其他绑定 2"-->
                <ListBoxItem Foreground="Red">其他绑定 2</ListBoxItem>
            </CompositeCollection>
        </ListBox.ItemsSource><!--ListBox 控件数据源绑定结束-->
    </ListBox>
</StackPanel>

```



```

</ListBox><!--ListBox 控件结束-->
</StackPanel><!-- StackPanel 控件结束-->
</Window><!-- Window 窗体结束-->

```

从以上代码可以看出，数据展现控件使用的是 ListBox 控件，该控件中的 ItemSource 属性通过 CompositeCollection 属性将 3 种数据源结合起来，并一起在 ListBox 中显示。第 1 部分为静态资源中 Source1Data 数据源。第 2 部分为静态资源中的 Source2Data 数据源。第 3 部分进行了数据的直接绑定。

(5) 添加后台 cs 代码。打开 MainWindow.xaml.cs 文件，在其中添加如下代码。

```

public class SourceItem
{
    //定义私有名称字段
    private string _name;
    //定义公共 Name 属性
    public string Name
    {
        //读取 Name 属性
        get
        {
            return _name;
        }
        //设定 Name 属性
        set
        {
            _name = value;
        }
    }
    //定义 SourceItem 属性
    public SourceItem(string name)
    {
        Name = name;
    }
}

//定义 Source1 数据源类
public class Source1 : ObservableCollection<SourceItem>
{
    //构造函数
    public Source1()
    {
        //添加数据项
        Add(new SourceItem("Source1 选项 1"));
        Add(new SourceItem("Source1 选项 2"));
        Add(new SourceItem("Source1 选项 3"));
        Add(new SourceItem("Source1 选项 4"));
        Add(new SourceItem("Source1 选项 5"));
    }
}

```

以上代码中定义了 Source1 集合类和集合中的对象类 SourceItem 类，这两个类全部用于前台进行数据源绑定的对象说明类。Source1 类继承了 ObservableCollection 类。ObservableCollection 类表示一个动态数据集合，在添加项、移除项或刷新整个列表时，此集合将提供通知。Add 方法表示向该 Source1 集合中添加数据对象。

(6) 运行该实例，效果如图 27.17 所示。



图 27.17 多数据源绑定效果

从图中可以看出，不同颜色代表不同数据源的绑定结果。

27.3.6 属性变更引起依赖数据绑定变化

当数据控件所绑定的属性发生变更的时候，可以通过事件的触发机制完成数据的重新绑定。本节将通过一个实例，介绍如何实现属性变更引起的数据绑定变化。

(1) 创建 WPF 应用程序，项目名称为 PropertyChangeSample。

(2) 打开 MainWindow.xaml 文件的 XAML 视图编辑，在页面中添加如下代码：

```
<!--定义 Window 窗体-->
<Window x:Class="PropertyChangeSample.MainWindow"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    <!--定义窗体的类名称-->
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    <!--定义 xmlns:x 命名空间-->
    xmlns:c="clr-namespace:PropertyChangeSample"
    <!--定义 xmlns:c 命名空间-->
    Title="数据实时跟踪" Height="300" Width="383">
    <!--定义窗体的标题,高度和宽度-->
    <!--添加 DockPanel 控件,同时设定背景色为 Cornsilk ,水平左对齐,垂直居上,宽度,
    高度和名称-->
    <DockPanel Background="Cornsilk" HorizontalAlignment="Left"
        VerticalAlignment="Top" Width="350" Height="150" Name="Page1">
    <!--添加 DockPanel 资源-->
    <DockPanel.Resources>
    <!--指定数据类和键值-->
    <c:StockCollection x:Key="MyDataProvider"/>
    <!--添加数据模板-->
    <DataTemplate x:Key="StockItemDataTemplate">
    <!--定义画布控件的宽度和高度-->
    <Canvas Width="300" Height="20">
    <!--添加 TextBlock 控件,指定字体大小 10,前景色 Blue,宽度 180,画布
```



```

        位置, 文本绑定属性-->
        <TextBlock FontSize="10" Foreground="Blue" Width="180"
        Canvas.Left="0" Text="{Binding Path=StockItemName}"/>
        <!--添加 TextBlock 控件, 指定字体大小 10, 前景色 Green , 文本为$,
        画布位置-->
        <TextBlock FontSize="10" Foreground="Green" Text="$"
        Canvas.Left="180"/>
        <!--添加 TextBlock 控件, 指定字体大小 10, 前景色 Green, 宽度 80, 画
        布位置, 文本绑定属性-->
        <TextBlock FontSize="10" Foreground="Green" Width="80"
        Canvas.Left="190" Text="{Binding Path=StockItemPrice}"/>
    </Canvas>
</DataTemplate>
</DockPanel.Resources>
<!--添加 TextBlock 控件, 指定字体大小 18, 边距 5, 字重 Bold, 在 DockPanel 的停
靠位置为上部 -->
<TextBlock FontSize="18" Margin="5" FontWeight="Bold"
DockPanel.Dock="Top">自选股票查询系统</TextBlock>
<!--添加 ListBox 控件, 名称为 MyListBox , 在 DockPanel 的停靠位置为上部, 宽度
315, 高度 80, 背景色 HoneyDew -->
<ListBox Name="MyListBox" DockPanel.Dock="Top" Width="315"
Height="80" Background="HoneyDew"
    <!--定义 ListBox 控件的子项数据源-->
    ItemsSource="{Binding Source={StaticResource
    MyDataProvider}}"
    <!--定义 ListBox 控件的子项模板-->
    ItemTemplate="{StaticResource StockItemDataTemplate}"/>
</DockPanel>
</Window>

```

在 XAML 代码中, 首先定义了 DockPanel 的资源文件, 在资源文件中定义了 ListBox 子项的展现风格, 同时指明了数据源的对象格式为 StockCollection。下面添加了一个 ListBox 控件实现最终数据的展现。

(3) 添加后台 cs 代码。打开 MainWindow.xaml.cs 文件, 在其中添加如下代码。

```

public class Stock : INotifyPropertyChanged
{
    //定义私有 _stockitemname 字段
    private string _stockitemname = "Unset";
    //定义私有 _stockitemprice 字段
    private decimal _stockitemprice = (decimal)0;
    //构造函数
    public Stock(string newStockItemName, decimal newStockItemPrice)
    {
        _stockitemname = newStockItemName;
        _stockitemprice = newStockItemPrice;
    }
    //定义 StockItemName 属性
    public string StockItemName
    {
        //获取 StockItemName 属性
        get
        {
            return _stockitemname;
        }
        //设置 StockItemName 属性
        set
    }
}

```



```

    {
        //判断 stockitemname 的值
        if (_stockitemname.Equals(value) == false)
        {
            _stockitemname = value;
            // 当 StockItemName 属性发生变化的时候, 调用 OnPropertyChanged
            // 事件
            OnPropertyChanged("StockItemName");
        }
    }
}
//定义 StockItemPrice 属性
public decimal StockItemPrice
{
    //获取 StockItemPrice 属性
    get
    {
        return _stockitemprice;
    }
    //设置 StockItemPrice 属性
    set
    {
        if (_stockitemprice.Equals(value) == false)
        {
            _stockitemprice = value;
            //当 StockItemPrice 属性发生变化的时候, 调用 OnPropertyChanged
            //事件
            OnPropertyChanged("StockItemPrice");
        }
    }
}
//定义属性变化事件委托
public event PropertyChangedEventHandler PropertyChanged;
//定义属性变更私有方法
private void OnPropertyChanged(string propName)
{
    if (PropertyChanged != null)
    {
        //调用委托事件
        PropertyChanged(this, new
            PropertyChangedEventArgs(propName));
    }
}
}
//定义数据源对象类
public class StockCollection : ObservableCollection<Stock>
{
    //添加数据源中的对象
    private Stock item1 = new Stock("科技股票", (decimal)24.95);
    private Stock item2 = new Stock("钢铁股票", (decimal)16.05);
    private Stock item3 = new Stock("银行股票", (decimal)100.0);
    //定义计时器事件
    private void Timer1_Elapsed(object sender, System.Timers.
        ElapsedEventArgs e)
    {
        //分别设定子项的 StockItemPrice 属性
        item1.StockItemPrice += (decimal)1.25;
        item2.StockItemPrice += (decimal)2.45;
    }
}

```




```

        item3.StockItemPrice += (decimal)10.55;
    }
    //创建时间私有方法
    private void CreateTimer()
    {
        //定义计时器对象
        System.Timers.Timer Timer1 = new System.Timers.Timer();
        //设定计时器为可用
        Timer1.Enabled = true;
        //设定计时器的时间间隔
        Timer1.Interval = 2000;
        //设定计时器的 Elapsed 事件
        Timer1.Elapsed += new System.Timers.ElapsedEventHandler
            (Timer1_Elapsed);
    }
    //定义构造函数
    public StockCollection()
        : base()
    {
        //添加子项
        Add(item1);
        Add(item2);
        Add(item3);
        //调用创建时间私有方法
        CreateTimer();
    }
}

```

在上面代码中，首先定义了用于显示的数据项类 `Stock`，其实现了 `InotifyPropertyChanged` 事件通知接口。该接口提供了 `PropertyChanged` 的事件，通过该事件的定义，可以完成向客户端发出某一属性值已更改的通知。为了实现接口，代码中定义了事件委托对象 `PropertyChanged`，并在 `StockItemName` 属性和 `StockItemPrice` 属性发生更新的时候，调用 `OnPropertyChanged` 方法来实现 `PropertyChanged` 委托的调用。

 **注意：**代码最后的 `StockCollection` 类定义，是一个数据绑定的集合类，通过该类最终构造了数据控件显示的内容。

(4) 运行该实例，效果如图 27.18 所示。



图 27.18 属性更新的数据绑定

运行结果会随着时间变化，显示不同时刻的属性结果，从而实现数据的不同绑定。

27.3.7 使用 Brush 填充图形

WPF 图形处理中,需要使用画笔将颜色或者图形、图片风格添加到页面中,这就需要 Brush 对象来实现。本节实例中将涉及 3 种 Brush 对象,分别是 SolidColorBrush、LinearGradientBrush 和 RadialGradientBrush。SolidColorBrush 使用纯色绘制区域;LinearGradientBrush 使用线性渐变封装 Brush;RadialGradientBrush 使用径向渐变绘制区域,焦点定义渐变的开始,而圆定义渐变的终点。3 种绘制方式各有不同,下面通过实例来分别介绍其绘制过程。

(1) 创建 WPF 浏览器应用程序,项目名称为 BrushSample。

(2) 打开 Page1.xaml 文件的 XAML 视图编辑,在页面中首先添加如下代码:

```
<!--定义 Page 页面资源-->
<Page.Resources>
    <!--定义 Rectangle 的样式属性-->
    <Style TargetType="{x:Type Rectangle}">
        <!--定义 Rectangle 的 Stroke 属性-->
        <Setter Property="Stroke" Value="Black"/>
        <!--定义 Rectangle 的 StrokeThickness 属性-->
        <Setter Property="StrokeThickness" Value="2"/>
        <!--定义 Rectangle 的 Margin 属性-->
        <Setter Property="Margin" Value="0,0,5,5"/>
        <!--定义 Rectangle 的 VerticalAlignment 属性-->
        <Setter Property="VerticalAlignment" Value="Top"/>
    </Style>
    <!--定义 Ellipse 样式属性-->
    <Style TargetType="{x:Type Ellipse}">
        <!--定义 Ellipse 的 Stroke 属性-->
        <Setter Property="Stroke" Value="Black"/>
        <!--定义 Ellipse 的 StrokeThickness 属性-->
        <Setter Property="StrokeThickness" Value="2"/>
        <!--定义 Ellipse 的 Margin 属性-->
        <Setter Property="Margin" Value="0,0,5,5"/>
    </Style>
</Page.Resources>
```

该段代码定义了页面的风格,目标控件为 Rectangle 和 Ellipse。

(3) 添加布局代码。接下来添加页面布局的代码如下:

```
<!--添加网格控件,指定网格控件的边距和背景色-->
<Grid Margin="10" Background="White">
    <!--添加网格列集合定义-->
    <Grid.ColumnDefinitions>
        <!--分别定义网格的各个列的宽度,设定为自动或者具体数值-->
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="10" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="10" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="10" />
    </Grid.ColumnDefinitions>
    <!--添加网格行集合定义-->
    <Grid.RowDefinitions>
```



```

<!--分别定义网格的各个行的高度,设定为自动或者具体数值-->
<RowDefinition Height="Auto" />
<RowDefinition Height="40" />
<RowDefinition Height="Auto" />
<RowDefinition Height="Auto" />
<RowDefinition Height="10" />
<RowDefinition Height="Auto" />
<RowDefinition Height="Auto" />
<RowDefinition Height="10" />
<RowDefinition Height="Auto" />
</Grid.RowDefinitions>

```

在界面中划分好各个区域,然后将要绘制的各个图形放置到对应的网格控件单元格中。

(4) 添加 `SolidColorBrush` 绘图控件。下面添加 `SolidColorBrush` 的绘制代码,如下所示。

```

<!-- SolidColorBrush 实例 -->
<!-- TextBlock 控件,第 2 行 0 列,内部文本为 SolidColorBrush -->
<TextBlock Grid.Row="2" Grid.Column="0">SolidColorBrush
</TextBlock>
<!--定义一个网格控件,位置为父网格控件的 3 行 0 列-->
<Grid Grid.Row="3" Grid.Column="0" >
    <!--添加网格列集合定义-->
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <!--添加网格行集合定义-->
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <!--添加矩形控件,指定填充颜色,宽度,高度以及在网格中的位置属性-->
    <Rectangle Fill="Red" Width="150" Height="150" Grid.Row="0"
    Grid.Column="0" Grid.RowSpan="3" />
    <!--添加矩形控件,指定填充颜色,宽度,高度以及在网格中的位置属性-->
    <Rectangle Fill="MediumBlue" Width="50" Height="50" Grid.Row="0"
    Grid.Column="1" />
    <!--添加矩形控件,指定填充颜色,宽度,高度以及在网格中的位置属性-->
    <Rectangle Fill="Purple" Width="50" Height="50" Grid.Row="1"
    Grid.Column="1" />
    <!--添加矩形控件,指定填充颜色,宽度,高度以及在网格中的位置属性-->
    <Rectangle Fill="Gold" Width="50" Height="50" Grid.Row="2"
    Grid.Column="1" />
</Grid>

```

在以上代码中,一共绘制了 4 个 `Rectangle` 图形,这些图形分别使用 `Fill` 属性填充了颜色,同时指定了每个 `Rectangle` 图形的宽度、高度,以及在网格中的位置等属性。

(5) 添加 `LinearGradientBrush` 绘图控件。添加 `LinearGradientBrush` 的绘制代码如下:

```

<!-- LinearGradientBrush 实例 -->
<!-- TextBlock 控件,第 2 行 2 列,内部文本为 LinearGradientBrush -->
<TextBlock Grid.Row="2" Grid.Column="2">LinearGradientBrush
</TextBlock>
<!--定义一个网格控件,位置为父网格控件的 3 行 2 列-->
<Grid Grid.Row="3" Grid.Column="2" >

```



```

<!--添加网格行集合定义-->
<Grid.ColumnDefinitions>
<ColumnDefinition />
<ColumnDefinition />
<ColumnDefinition />
</Grid.ColumnDefinitions>
<!--添加网格行集合定义-->
<Grid.RowDefinitions>
<RowDefinition />
<RowDefinition />
<RowDefinition />
</Grid.RowDefinitions>
<!--添加矩形控件,指定宽度 150,高度 150.以及在网格中的位置为 0 行 0 列-->
<Rectangle Width="150" Height="150" Grid.Row="0" Grid.Column="0"
Grid.RowSpan="3">
<!-- 定义 Rectangle 的填充属性 -->
<Rectangle.Fill>
<!-- 定义 LinearGradientBrush 的 GradientStops 属性 -->
<LinearGradientBrush>
<!--描述 LinearGradientBrush 渐变中过渡点和颜色属性集合-->
<LinearGradientBrush.GradientStops>
<!--描述渐变中过渡点的位置为 0.0 和颜色为 Yellow -->
<GradientStop Color="Yellow" Offset="0.0" />
<!--描述渐变中过渡点的位置为 0.5 和颜色为 Orange -->
<GradientStop Color="Orange" Offset="0.5" />
<!--描述渐变中过渡点的位置为 1.0 和颜色为 Red -->
<GradientStop Color="Red" Offset="1.0" />
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
<!--添加矩形控件,指定宽度 50,高度 50.以及在网格中的位置为 0 行 1 列-->
<Rectangle Width="50" Height="50" Grid.Row="0" Grid.Column="1">
<!-- 定义 Rectangle 的填充属性 -->
<Rectangle.Fill>
<!-- 定义 LinearGradientBrush -->
<LinearGradientBrush>
<!-- 定义 LinearGradientBrush 的 GradientStops 属性 -->
<LinearGradientBrush.GradientStops>
<!--描述渐变中过渡点的位置为 0.0 和颜色为 Blue -->
<GradientStop Color="Blue" Offset="0.0" />
<!--描述渐变中过渡点的位置为 1.0 和颜色为 Purple -->
<GradientStop Color="Purple" Offset="1.0" />
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
<!--添加矩形控件,指定宽度 50,高度 50.以及在网格中的位置为 1 行 1 列-->
<Rectangle Width="50" Height="50" Grid.Row="1" Grid.Column="1">
<!-- 定义 Rectangle 的填充属性 -->
<Rectangle.Fill>
<!-- 定义 LinearGradientBrush -->
<LinearGradientBrush>
<!--描述 LinearGradientBrush 渐变中过渡点和颜色属性集合-->
<LinearGradientBrush.GradientStops>
<!--描述渐变中过渡点的位置为 0.0 和颜色为 Purple -->
<GradientStop Color="Purple" Offset="0.0" />
<!--描述渐变中过渡点的位置为 0.5 和颜色为 BlueViolet -->

```



```

        <GradientStop Color="BlueViolet" Offset="0.5" />
        <!--描述渐变中过渡点的位置为 1.0 和颜色为 White -->
        <GradientStop Color="White" Offset="1.0" />
    </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
<!--添加矩形控件,指定宽度 50,高度 50.以及在网格中的位置为 2 行 1 列-->
<Rectangle Width="50" Height="50" Grid.Row="2" Grid.Column="1">
    <!-- 定义 Rectangle 的填充属性 -->
    <Rectangle.Fill>
        <!-- 定义 LinearGradientBrush -->
        <LinearGradientBrush>
            <!--描述 LinearGradientBrush 渐变中过渡点和颜色属性集合-->
            <LinearGradientBrush.GradientStops>
                <!--描述渐变中过渡点的位置为 0.0 和颜色为 Gold -->
                <GradientStop Color="Gold" Offset="0.0" />
                <!--描述渐变中过渡点的位置为 0.5 和颜色为 Red -->
                <GradientStop Color="Red" Offset="0.5" />
                <!--描述渐变中过渡点的位置为 1.0 和颜色为 Orange -->
                <GradientStop Color="Orange" Offset="1.0" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle> <!-- Rectangle 控件结束-->
</Grid> <!-- Grid 控件结束-->

```

在上面代码中,同样绘制了 4 个 Rectangle 图形,每个图形同样使用了 Fill 属性进行颜色的填充。不同的是添加了 LinearGradientBrush 属性集合,并且通过 GradientStops 属性来完成颜色的渐变过程。

(6) 添加 RadialGradientBrush 绘图控件。添加 RadialGradientBrush 的绘制代码如下:

```

<!-- RadialGradientBrush 实例. -->
<!--TextBlock 控件,第 2 行 4 列,内部文本为 RadialGradientBrush -->
<TextBlock Grid.Row="2" Grid.Column="4">RadialGradientBrush
</TextBlock>
<!--定义一个网格控件,位置为父网格控件的 3 行 4 列-->
<Grid Grid.Row="3" Grid.Column="4">
    <!--添加网格行集合定义-->
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <!--添加网格行集合定义-->
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <!--添加矩形控件,指定宽度 150,高度 150.以及在网格中的位置为 0 行 0 列-->
    <Rectangle Width="150" Height="150" Grid.Row="0" Grid.Column="0"
    Grid.RowSpan="3">
        <!-- 定义 Rectangle 的填充属性 -->
        <Rectangle.Fill>
            <!-- 定义 RadialGradientBrush 的起始位置为 0.75,0.25 -->
            <RadialGradientBrush GradientOrigin="0.75,0.25">

```



```

<!--描述 RadialGradientBrush 渐变中过渡点和颜色属性集合-->
<RadialGradientBrush.GradientStops>
  <!--描述渐变中过渡点的位置为 0.0 和颜色为 Yellow -->
  <GradientStop Color="Yellow" Offset="0.0" />
  <!--描述渐变中过渡点的位置为 0.5 和颜色为 Orange -->
  <GradientStop Color="Orange" Offset="0.5" />
  <!--描述渐变中过渡点的位置为 1.0 和颜色为 Red -->
  <GradientStop Color="Red" Offset="1.0" />
</RadialGradientBrush.GradientStops>
</RadialGradientBrush>
</Rectangle.Fill>
</Rectangle>
<!--添加椭圆形控件,指定宽度 50,高度 50.以及在网格中的位置为 0 行 1 列-->
<Ellipse Width="50" Height="50" Grid.Row="0" Grid.Column="1">
  <!-- 定义 Ellipse 的填充属性 -->
  <Ellipse.Fill>
    <!-- 定义 RadialGradientBrush -->
    <!-- 定义 RadialGradientBrush 的起始位置为 0.75,0.25 -->
    <RadialGradientBrush GradientOrigin="0.75,0.25">
      <!--描述 RadialGradientBrush 渐变中过渡点和颜色属性集合-->
      <RadialGradientBrush.GradientStops>
        <!--描述渐变中过渡点的位置为 0.0 和颜色为 White -->
        <GradientStop Color="White" Offset="0.0" />
        <!--描述渐变中过渡点的位置为 0.5 和颜色为 MediumBlue -->
        <GradientStop Color="MediumBlue" Offset="0.5" />
        <!--描述渐变中过渡点的位置为 1.0 和颜色为 Black -->
        <GradientStop Color="Black" Offset="1.0" />
      </RadialGradientBrush.GradientStops>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
<!--添加椭圆形控件,指定宽度 50,高度 50.以及在网格中的位置为 1 行 1 列-->
<Ellipse Width="50" Height="50" Grid.Row="1" Grid.Column="1">
  <!-- 定义 Ellipse 的填充属性 -->
  <Ellipse.Fill>
    <!-- 定义 RadialGradientBrush -->
    <!-- 定义 RadialGradientBrush 的起始位置为 0.75,0.25 -->
    <RadialGradientBrush GradientOrigin="0.75,0.25">
      <!--描述 RadialGradientBrush 渐变中过渡点和颜色属性集合 -->
      <RadialGradientBrush.GradientStops>
        <!--描述渐变中过渡点的位置为 0.0 和颜色为 AliceBlue -->
        <GradientStop Color="AliceBlue" Offset="0.0" />
        <!--描述渐变中过渡点的位置为 0.5 和颜色为 Purple -->
        <GradientStop Color="Purple" Offset="0.5" />
        <!--描述渐变中过渡点的位置为 1.0 和颜色为 #330033-->
        <GradientStop Color="#330033" Offset="1.0" />
      </RadialGradientBrush.GradientStops>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
<!--添加椭圆形控件,指定宽度 50,高度 50.以及在网格中的位置为 2 行 1 列-->
<Ellipse Width="50" Height="50" Grid.Row="2" Grid.Column="1">
  <!-- 定义 Ellipse 的填充属性 -->
  <Ellipse.Fill>
    <!-- 定义 RadialGradientBrush -->
    <!-- 定义 RadialGradientBrush 的起始位置为 0.75,0.25 -->
    <RadialGradientBrush GradientOrigin="0.75,0.25">

```



```

<!--描述 RadialGradientBrush 渐变中过渡点和颜色属性集合-->
<RadialGradientBrush.GradientStops>
    <!--描述渐变中过渡点的位置为 0.0 和颜色为 Yellow -->
    <GradientStop Color="Yellow" Offset="0.0" />
    <!--描述渐变中过渡点的位置为 0.5 和颜色为 Orange -->
    <GradientStop Color="Orange" Offset="0.5" />
    <!--描述渐变中过渡点的位置为 1.0 和颜色为 Red -->
    <GradientStop Color="Red" Offset="1.0" />
</RadialGradientBrush.GradientStops>
</RadialGradientBrush>
</Ellipse.Fill>
</Ellipse> <!-- Ellipse 控件结束-->
</Grid> <!-- Grid 控件结束-->

```

上面的代码绘制了 4 个 Ellipse 图形，同样使用 Fill 属性填充颜色。但由于使用的是 RadialGradientBrush 进行填充，所以在 Fill 属性中，添加了 RadialGradientBrush 的集合，同时指定了每个 Ellipse 图形的 GradientStops 属性来设置颜色的渐变。

(7) 运行该实例，效果如图 27.19 所示。

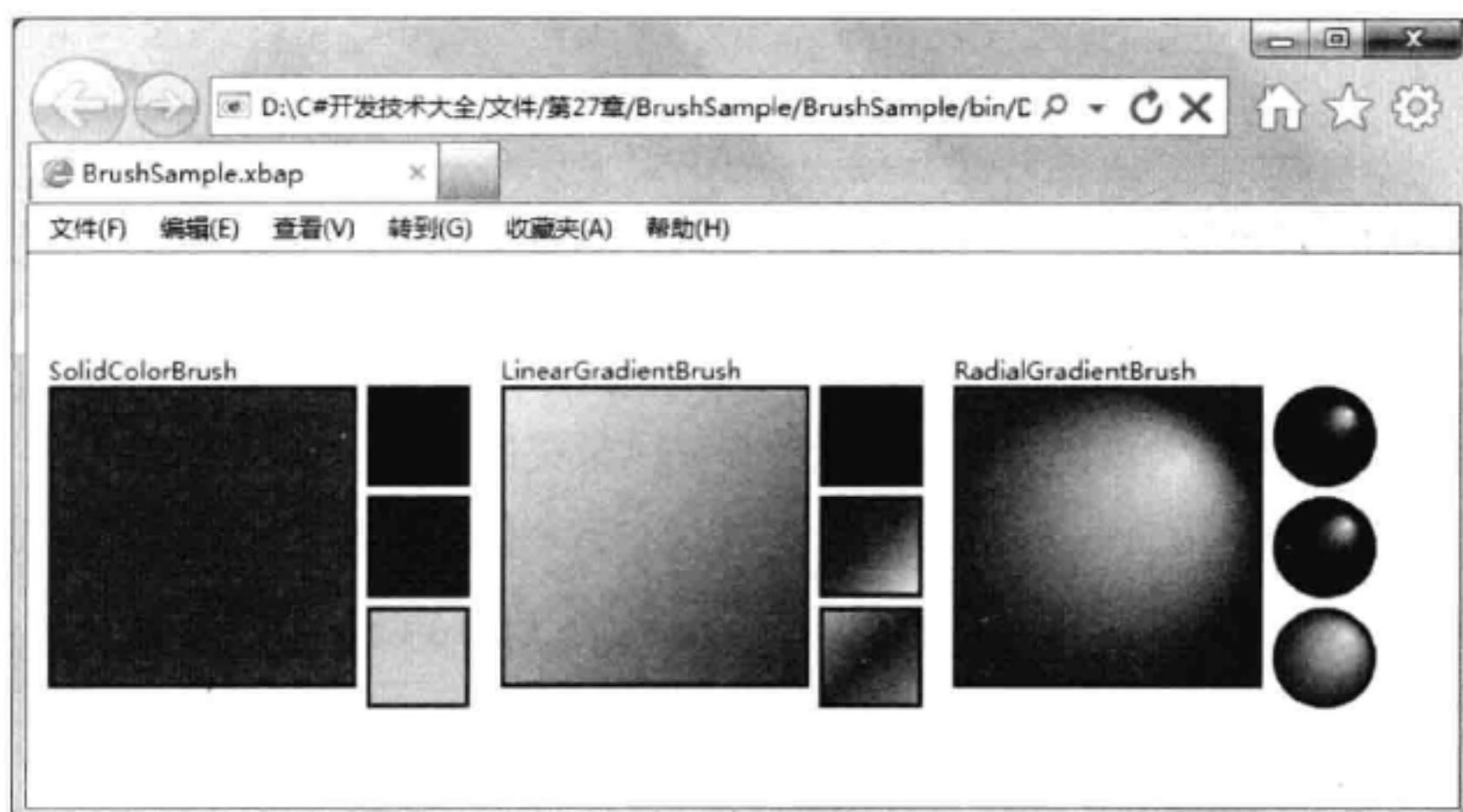


图 27.19 使用 Brush 填充图形效果

27.3.8 使用 Storyboard 实现动画

在 WPF 中使用动画，可以使控件和元素变大、晃动、旋转和淡化，还可以产生有趣的页面过渡和更多的效果。WPF 允许对大多数属性进行动画处理。

说明：不但可以对大多数 WPF 对象进行动画处理，而且还可以使用 WPF 对创建的自定义对象进行动画处理。

本节将通过一个实例来介绍如何使用 WPF 的 Storyboard 实现控制图形尺寸，形成动画的实例。

(1) 创建 WPF 应用程序，项目名称为 AnimatedSample。

(2) 打开 MainWindow.xaml 文件的 XAML 视图编辑，在页面中首先添加如下代码：

```
<DockPanel>
```

```

<!--定义用于动画的矩形控件,宽度为 200,高度为 200,名称为 myRectangle -->
<Rectangle Width="200" Height="200" Name="myRectangle">
  <!--填充矩形控件内容-->
  <Rectangle.Fill>
    <!--定义 LinearGradientBrush 的起始点,扩展方式为渐变向量末端的颜色值填充剩余的空间,终止点,以及映射方式-->
    <LinearGradientBrush StartPoint="0,0" SpreadMethod="Pad" EndPoint="0,1" MappingMode="RelativeToBoundingBox">
      <!--定义填充控件属性-->
      <LinearGradientBrush.GradientStops>
        <!--描述 LinearGradientBrush 渐变中过渡点和颜色属性集合 -->
        <GradientStopCollection>
          <!--描述渐变中过渡点的位置为 0 和颜色为 red -->
          <GradientStop Offset="0" Color="red" />
          <!--描述渐变中过渡点的位置为 0.5 和颜色为 green -->
          <GradientStop Offset="0.5" Color="green" />
          <!--描述渐变中过渡点的位置为 0.9074074 和颜色为 blue-->
          <GradientStop Offset="0.9074074" Color="blue" />
        </GradientStopCollection>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush> <!-- LinearGradientBrush 控件结束-->
  </Rectangle.Fill> <!-- Fill 控件结束-->
</Rectangle> <!-- Rectangle 控件结束-->

```

本段 XAML 代码定义了一个 Rectangle 图形,同时使用了 LinearGradientBrush 对其进行颜色的填充。之所以定义这个图形,是为了后续对其部分属性进行修改,从而实现动画的效果。

(3) 添加动画效果代码。接下来添加图形动画效果的代码,如下所示。

```

<DockPanel.Triggers>
  <!--定义 DockPanel 事件触发器-->
  <EventTrigger RoutedEvent="Page.Loaded">
    <!--定义开始动画板的名称为 myBeginStoryboard -->
    <BeginStoryboard Name="myBeginStoryboard">
      <!--定义动画板的名称为 myStoryboard -->
      <Storyboard Name="myStoryboard">
        <!--定义动画处理对象,指定动画板目标,动画属性为矩形的高度,动画的起止范围,重复时间属性,开始时间属性 -->
        <DoubleAnimation Storyboard.TargetName="myRectangle" Storyboard.TargetProperty="(Rectangle.Height)" From="0" To="100" RepeatBehavior="0:0:50" BeginTime="0:0:0.5" />
        <!--定义动画处理对象,指定动画板目标,动画属性为矩形的高度,动画的起止范围,是否自动翻转,重复间隔,开始时间属性以及动画的持续时间 -->
        <DoubleAnimation Storyboard.TargetName="myRectangle" Storyboard.TargetProperty="(Rectangle.Height)" From="0" To="100" AutoReverse="true" RepeatBehavior="0:0:50" BeginTime="0:0:0.5" Duration="0:0:2"/>
      </Storyboard>
    </BeginStoryboard> <!-- BeginStoryboard 控件结束-->
  </EventTrigger> <!-- EventTrigger 控件结束-->
</DockPanel.Triggers> <!-- Triggers 控件结束-->
</DockPanel> <!-- DockPanel 控件结束-->

```

在以上代码中,首先定义了触发动画开始的事件为 Page.Loaded,同时在

BeginStoryboard 中定义了 Storyboard 对象,然后在 Storyboard 中定义了 DoubleAnimation 对象。BeginStoryboard 类是一个触发器操作,该操作可启动 Storyboard 并将其动画分发给动画的目标对象和属性。Storyboard 类为容器的子动画提供对象和属性目标信息的容器时间线。DoubleAnimation 对象在指定的 Duration 内使用线性内插对两个目标值之间的 Double 属性值进行动画处理。DoubleAnimation 对象的各个属性如表 27.2 所示。

表 27.2 DoubleAnimation对象属性说明

名 称	说 明
TargetName	目标图形名称
TargetProperty	需要变化的目标图形属性
From	获取或设置动画的起始值
To	获取或设置动画的结束值
AutoReverse	获取或设置一个值,该值指示时间线在完成向前迭代后是否按相反的顺序播放
RepeatBehavior	获取或设置此时间线的重复行为
BeginTime	获取或设置此Timeline将要开始的时间
Duration	获取或设置此时间线播放的时间长度

(4) 运行该实例,效果如图 27.20 所示。



图 27.20 使用 Storyboard 实现动画效果

可以看出,该图形随着事件的变化,高度也在不停地变化。

27.3.9 使用 Storyboard 实现控件的翻转

除了可以通过对图形的某一属性控制获取动画效果之外,还可以通过设定控件在多个关键帧的不同属性来完成控件的动画效果。本节将通过实例介绍如何使用关键帧技术,实现按钮的翻转效果。

- (1) 创建 WPF 浏览器应用程序,项目名称为 RotationSample。
- (2) 打开 Page1.xaml 文件的 XAML 视图编辑,在页面中添加如下代码:

```
<StackPanel Orientation="Vertical" HorizontalAlignment="Center">
    <!--定义旋转按钮的名称,边距,宽度,提交变形的起止位置-->
    <Button Name="myAnimatedButton" Margin="200" Width="120"
        RenderTransformOrigin="0.5,0.5">单击之后开始旋转
```

```

<!--定义旋转按钮的变化目标为角度-->
<Button.RenderTransform>
    <RotateTransform Angle="0" />
</Button.RenderTransform>
<!--定义按钮触发器-->
<Button.Triggers>
    <!--动画开始触发时间为单击按钮-->
    <EventTrigger RoutedEvent="Button.Click">
        <!--定义开始动画板对象-->
        <BeginStoryboard>
            <!--定义动画板对象-->
            <Storyboard>
                <!--定义动画对象 DoubleAnimationUsingKeyFrames , 动画目标控件名称, 目标属性名称以及控制动画的控件属性, 持续时间和填充属性-->
                <DoubleAnimationUsingKeyFrames Storyboard.TargetName="myAnimatedButton" Storyboard.TargetProperty="(Button.RenderTransform).(RotateTransform.Angle)" Duration="0:0:10" FillBehavior="Stop">
                    <!--定义动画帧集合-->
                    <DoubleAnimationUsingKeyFrames.KeyFrames>
                        <!--控件角度为 300 度, 时间点为第 3 秒-->
                        <LinearDoubleKeyFrame Value="300" KeyTime="0:0:3" />
                        <!--控件角度为 225 度, 时间点为第 3.5 秒-->
                        <DiscreteDoubleKeyFrame Value="225" KeyTime="0:0:3.5" />
                        <!--控件角度为 180 度, 时间点为第 4 秒-->
                        <DiscreteDoubleKeyFrame Value="180" KeyTime="0:0:4" />
                        <!--控件角度为 90 度, 时间点为第 4.5 秒-->
                        <DiscreteDoubleKeyFrame Value="90" KeyTime="0:0:4.5" />
                        <!--控件角度为 -180 度, 时间点为第 10 秒-->
                        <SplineDoubleKeyFrame Value="-180" KeyTime="0:0:10" KeySpline="0.25,0.5 0.75,1" />
                    </DoubleAnimationUsingKeyFrames.KeyFrames>
                </DoubleAnimationUsingKeyFrames>
            </Storyboard><!-- Storyboard 控件结束-->
        </BeginStoryboard><!-- BeginStoryboard 控件结束-->
    </EventTrigger><!-- EventTrigger 控件结束-->
</Button.Triggers><!-- Triggers 控件结束-->
</Button><!-- Button 控件结束-->
</StackPanel><!-- StackPanel 控件结束-->

```

在上面的代码中, 设定了按钮控件的 Triggers 属性, 表明按钮单击事件之后, 开始动画效果。DoubleAnimationUsingKeyFrames 属性中, 定义了该动画的目标控件名称、动画的目标属性、持续时间等内容。DoubleAnimationUsingKeyFrames.KeyFrames 的属性中, 定义了按钮在旋转过程中, 走到每个角度的过渡属性, 其中各个关键帧, 如 LinearDoubleKeyFrame、DiscreteDoubleKeyFrame 和 SplineDoubleKeyFrame 都是不同的帧类型。LinearDoubleKeyFrame 关键帧表示通过使用线性内插, 可以在前一个关键帧的 Double 值及其自己的 Value 之间进行动画处理。DiscreteDoubleKeyFrame 关键帧表示通过使用离散内插, 可以在前一个关键帧的 Double 值及其自己的 Value 之间进行动画处理。

SplineDoubleKeyFrame 关键帧表示通过使用样条内插,可以在前一个关键帧的 Double 值及其自己的 Value 之间进行动画处理。

(3) 运行该实例,效果如图 27.21 所示。



图 27.21 使用 Storyboard 实现控件的翻转效果

单击图上的按钮之后,该按钮开始随着时间进行旋转。

27.4 本章总结


本章介绍了.NET 框架中一个用于展现的系统框架 WPF。作为微软的下一代应用程序处理框架,WPF 无论从处理效果、开发效率和灵活性等方面,都展现出其强大的优势。本章通过对其基础架构的简单介绍,以及应用程序的实例介绍,使读者能够掌握 WPF 的基本概念和一般的 WPF 程序开发步骤。

27.5 实战练习

1. 在 Visual Studio 2010 中新建一个 WPF 应用程序,向 MainWindow 中添加一个 ListBox 控件和一个 Label 控件,向 ListBox 控件中添加一些选项,当选中某个选项时,该项的内容在 Label 控件中显示出来。

2. 在 Visual Studio 2010 中新建一个 WPF 应用程序,向 MainWindow 中添加一个 DockPanel 控件,在其中添加两个 TextBlock 控件,分别用来表示窗体的顶部和底部内容。

3. 在 Visual Studio 2010 中新建一个 WP 浏览器应用程序,用 StackPanel 控件设计一个用户注册面板。

 **提示:** 将标签、文本框等控件按从上到下的顺序放置在 StackPanel 中,再设置各控件的水平对齐方式即可。

4. 在 Visual Studio 2010 中新建一个 WP 浏览器应用程序,使用 Storyboard 控制一个水平渐变的动画效果。

第 28 章 WCF 框架

Windows Communication Foundation (WCF) 是新一代的消息通信框架，它提供了一组运行库和一组用于通信的 API 接口，这些组件可以创建客户端和服务端的消息通信机制，完成消息的传输。本章将介绍 WCF 架构及基础知识，然后通过搭建一个完整的服务框架，使读者更加深刻地理解 WCF 框架。

28.1 WCF 基础

WCF 提供的是一种通信的基础架构，它通过一种面向服务的新型编程模型简化了各个独立应用程序的开发。通过提供分层的体系结构，WCF 可以支持多种风格的分布式应用程序开发。建立在 WCF 基础之上的，是用于进行安全可靠的事务处理、数据交换的各种协议功能，以及广泛的传输协议和编码选择。

28.1.1 了解 WCF 架构

WCF 从架构上分为协定、服务运行时、消息传递，以及服务承载几个模块。各个模块的组成，以及包含的子组件如图 28.1 所示。

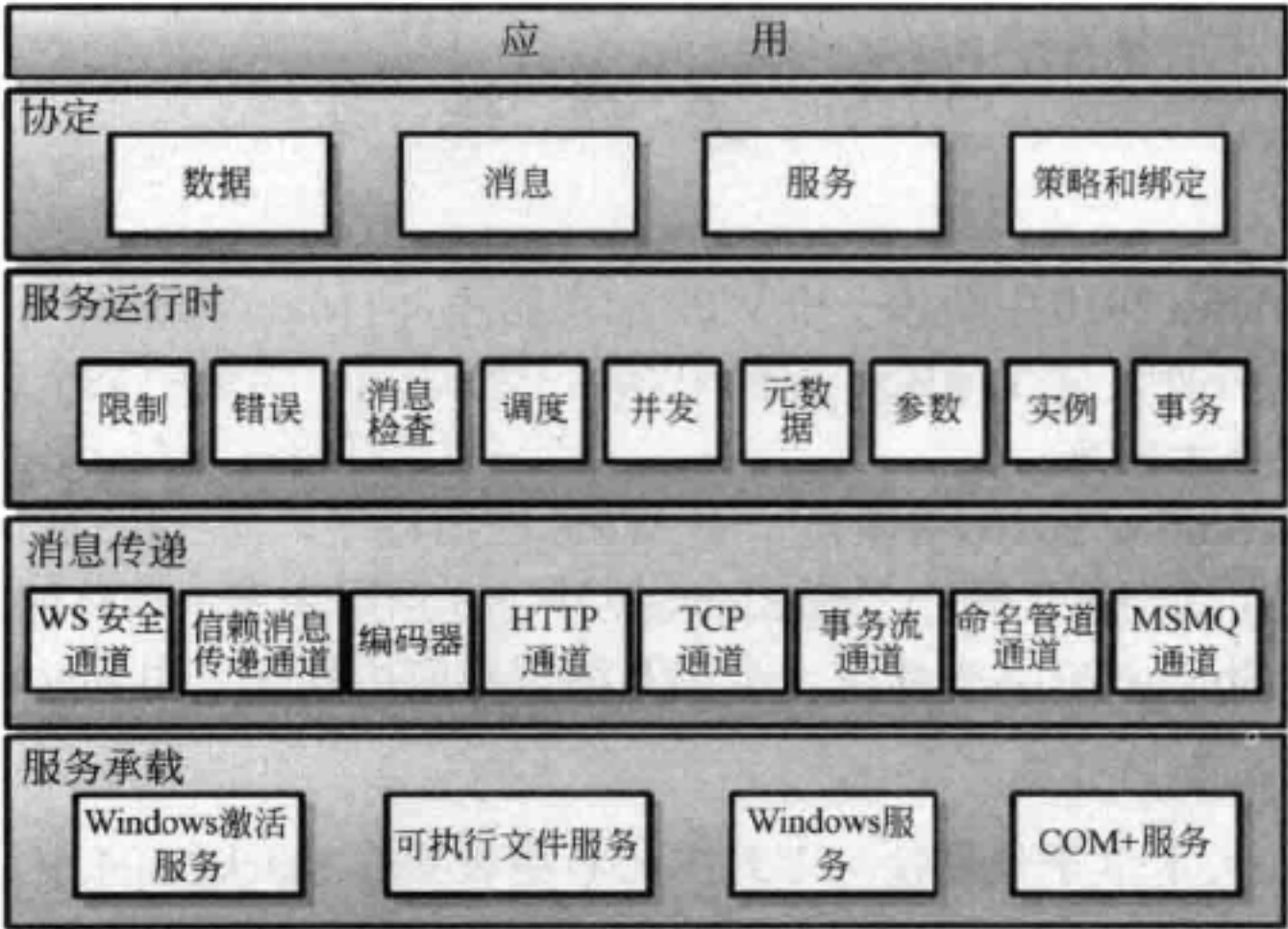



图 28.1 WCF 基础架构

- 协定：协定用于定义 WCF 框架之中的各个消息方面的规则。这些规则包括数据协定、消息协定、服务协定，以及策略和绑定之间的规则。数据协定用于定义每个


消息的每个参数。消息协定使用 SOAP 协议定义特定消息部分，当互操作性要求对消息的某些部分进行更精细的控制时，消息协定可实现这种控制。

 **说明：**服务协定指定服务的实际方法签名，并以支持的编程语言之一，作为接口进行分发。策略和绑定规定与某一服务进行通信所需的条件。

- **服务运行时：**服务运行时是指在 WCF 中的服务运行时的行为，它基本包括限制行为、错误行为、元数据行为、实例行为、事务行为、调度行为、开发行为。限制行为控制服务处理的消息数量。错误行为是指在服务出现内部错误的时候，需要采取的操作。元数据行为定义了如何提供给外界数据。实例行为定义了可以运行的实例数量。事务行为用来完成统一的原子事务操作，即在失败时回滚已进行事务处理的操作。调度行为用于控制 WCF 基础结构处理消息的方式。并发行为用来处理实例的并发控制。
- **消息传递：**消息传递由通道组成，通道是消息处理的组件。有两种类型的通道，包括传输通道和协议通道。传输通道包括 HTTP、命名管道、TCP 和 MSMQ。协议通道经常通过读取或写入消息的其他头的方式来实现消息处理协议。此类协议的示例包括 WS-Security 和 WS-Reliability。
- **服务承载：**服务承载层是指 WCF 运行的时候需要提供的服务类别。这些类别包括 Windows 激活服务、可执行文件、Windows 服务和 COM+ 服务。

28.1.2 了解 WCF 模型

WCF 服务模型以新的通信机制作为技术依托，并整合了现有的 ASP.NET Web 服务、.NET Framework 远程处理和企业服务等开发基础。该服务模型的特点在于，它将 Web 服务的概念直接映射到 .NET Framework 公共语言运行库中的对应内容，包括将消息灵活且可扩展地映射到用诸如 Visual C# 或 VB 等语言实现的服务。

 **说明：**该服务模型提供支持松散耦合和版本管理的序列化功能，并提供与诸如消息队列 (MSMQ)、COM+、ASP.NET Web 服务、Web 服务增强等现有 .NET Framework 分布式系统技术，以及很多其他功能的集成和互操作性。

下面介绍 WCF 模型中涉及的模块名称及说明。

- **消息：**应用与应用之间的传递信息单元，包括消息头和消息体等部分。
- **服务：**服务提供了终结点和服务操作的集合。一个服务公开了一个或多个终结点，每个终结点都公开一个或多个服务操作。
- **终结点：**终结点定义了地址、绑定和服务协定。
- **应用程序终结点：**由应用程序公开并对应于该应用程序实现的服务协定。
- **基础结构终结点：**由基础结构公开，以便实现与服务协定无关的服务需要或提供的功能。
- **地址：**消息可以发送到的目的地的位置。它以统一资源标识符 (URI) 的形式指定。
- **绑定：**定义终结点与外界进行通信的方式。它是消息应如何发送的通信机制规范。
- **绑定元素：**表示绑定的特定部分，如传输协议、编码、基础结构及协议。

- 行为：行为是控制服务、终结点、特定操作或客户端的各种运行时方面的要素。
- 服务操作：在服务的代码中定义的过程，用于实现某种操作的功能。
- 服务协定：描述了可以发送何种消息。
- 操作协定：定义参数并返回操作的类型。
- 消息协定：描述了消息的格式。
- 错误协定：可以将错误协定与服务操作进行关联，以指示可能返回到调用方的错误。
- 数据协定：数据类型的说明称为数据协定。
- 实例化：服务的具体实现形式，包括单个、每个调用和每个会话 3 种类型。单个类型，由单个程序对象为所有客户端提供服务；每个调用类型，创建一个新的程序对象来处理每个客户端调用；每个会话类型，创建一组程序对象，并且为每个独立的会话使用一个对象。
- 客户端应用程序：与一个或多个终结点交换消息的程序。
- 安全：WCF 中的安全包括保密性、完整性、身份验证。
- 传输安全模式：可以通过 3 种模式之一来保证安全，即传输模式、消息安全模式和使用消息凭据的传输模式。
- 消息安全模式：消息安全模式指定通过实现一个或多个安全规范来保证安全。

28.2 WCF 服务和客户端实例

WCF 定义了服务以及调用服务客户端的通信架构，所以创建一个完整的服务以及实现客户端对服务的调用，对于程序创建来说是最基本的。本节通过一个完整的应用，介绍如何创建、定义、实现和运行 WCF 服务，以及如何创建、配置和使用 WCF 客户端。

28.2.1 创建 WCF 服务承载项目

WCF 应用程序可以理解为承载服务的一类应用程序。在 Visual Studio 2010 中，它不是一个特殊的应用程序类型，可以通过最简单的控制台应用程序来创建这类程序，然后通过启动运行控制台应用程序在指定的地址和端口启动 WCF 服务，实现客户端对它的调用。本节实例创建的 WCF 就运行于控制台应用程序之上。

创建 WCF 服务，首先需要创建服务的承载项目。启动 Visual Studio 2010，选择菜单栏的“文件”|“新建”|“项目”命令，在弹出的“新建项目”对话框，在左侧的“已安装的模板”区域选择“Visual C#/Windows”选项，然后在右侧的“模板”区域选择“控制台应用程序”选项。在“名称”文本框中输入项目名称为 WCFService，在“位置”栏选择项目保存的路径，如图 28.2 所示。

28.2.2 定义 WCF 服务协定

创建 WCF 服务，首先需要定义与外界调用该服务的客户端的约定，该约定称为服务

协定。通过该协定的定义，可以描述如何与服务进行通信。在该约定中，需要定义输入和输出类型的接口。本节在 28.2.1 节创建的 WCF 服务承载项目的基础上，继续定义 WCF 的服务协定，具体步骤如下所述。



图 28.2 创建承载 WCF 服务的控制台应用程序

(1) 添加组件。在项目的“解决方案资源管理器”对话框中，右键单击项目名称 WCFService，在弹出的菜单中单击“添加引用”菜单。在弹出的“添加引用”对话框中，单击.NET 选项卡，然后确保选中“组件名称”为 System.ServiceModel 的组件之后，单击“确定”按钮，完成组件的引用，如图 28.3 所示。



图 28.3 添加 System.ServiceModel 组件

(2) 添加命名空间。双击项目中的 Program.cs 文件，在代码编辑区域的命名空间引用代码区域，添加新的命名空间的引用代码，如下所示。


```
using System.ServiceModel;
```

(3) 定义服务协定。服务协定是描述服务需要实现的一系列接口，接口中定义了服务

需要实现的方法，以及各个方法的属性描述。下面的代码描述了一个服务接口，并且定义了该接口对应的4个服务方法。

```
//定义服务协定。
[ServiceContract(Namespace = "http://VS2010.WCF.Sample")]
//服务接口
public interface IService
{
    //加操作方法
    [OperationContract]
    double Add(double n1, double n2);
    //减操作方法
    [OperationContract]
    double Subtract(double n1, double n2);
    //乘操作方法
    [OperationContract]
    double Multiply(double n1, double n2);
    //除操作方法
    [OperationContract]
    double Divide(double n1, double n2);
}
```

该服务接口的名称为 `IService`，通过接口定义的关键字 `Interface` 来实现。接口通过属性 `ServiceContract` 来标注该接口是服务协定，同时标注了该服务协定的命名空间为 `http://VS2010.WCF.Sample`。在该接口中，定义了 `Add`、`Subtract`、`Multiply` 和 `Divide` 等4个接口方法，每个方法都通过属性 `OperationContract` 进行描述，同时每个方法还定义了对应的输入和返回的参数类型。

 **说明：**通过以上几个步骤，完成了服务协定的定义。服务协定不包含具体的实现逻辑代码，但是却概括了该接口服务需要实现的功能，起到了协定双方的作用。

28.2.3 定义实现 WCF 服务接口的类

定义了 WCF 服务协定之后，需要定义一个实现该接口的类，在该类中将实现接口中定义的各个方法的内部逻辑。本节接着介绍如何实现 WCF 的服务类。

(1) 添加服务类。在 `Program.cs` 方法中，添加一个新的类 `CalculatorService`，同时使该类继承接口 `IService`。代码如下所示。

```
public class CalculatorService : IService
```

(2) 实现该类的接口方法。由于类 `CalculatorService` 继承了 `IService`，所以就需要实现接口中已经定义的方法。实现方法的具体代码如下所示。

```
//定义加服务操作
public double Add(double n1, double n2)
{
    //定义返回结果为输入参数之和
    double result = n1 + n2;
    //在控制台显示结果
    Console.WriteLine("调用方法 Add({0},{1})", n1, n2);
    Console.WriteLine("返回: {0}", result);
    return result;
}
```



```

    }
    //定义减服务操作
    public double Subtract(double n1, double n2)
    {
        //定义返回结果为输入参数之差
        double result = n1 - n2;
        //在控制台显示调用结果
        Console.WriteLine("调用方法 Subtract({0},{1})", n1, n2);
        Console.WriteLine("返回: {0}", result);
        return result;
    }
    //定义乘服务操作
    public double Multiply(double n1, double n2)
    {
        //定义返回结果为输入参数之积
        double result = n1 * n2;
        //在控制台显示调用结果
        Console.WriteLine("调用方法 Multiply({0},{1})", n1, n2);
        Console.WriteLine("返回: {0}", result);
        return result;
    }
    //定义除服务操作
    public double Divide(double n1, double n2)
    {
        //定义返回结果为输入参数之商
        double result = n1 / n2;
        //在控制台显示调用结果
        Console.WriteLine("调用方法 Divide({0},{1})", n1, n2);
        Console.WriteLine("返回: {0}", result);
        return result;
    }
}

```

到本节为止，WCF 服务的服务协定，以及具体的服务逻辑全部实现完毕。

28.2.4 运行 WCF 服务的相关代码

WCF 服务协定以及服务逻辑实现之后，如果客户端要对其进行调用，则需要将该服务协定运行起来，运行 WCF 就需要创建地址、创建服务终结点、激活元数据交换，以及最终的服务启动和停止的功能。本节将介绍如何通过程序实现运行 WCF 服务。

(1) 创建基本地址和服务主机。若要发布和启动服务，首先需要定义该服务运行的地址和端口。在类 Program 中的 Main 方法中，定义如下地址实例：

```

//创建基本地址
Uri baseAddress = new Uri("http://localhost:8000/ServiceModelSamples/Service");
//创建服务主机
ServiceHost selfHost = new ServiceHost(typeof(CalculatorService),
baseAddress);

```

该基本地址定义了服务的具体访问路径。

(2) 添加服务终结点，代码如下：

```


//创建服务终结点
selfHost.AddServiceEndpoint(

```



```
typeof(IService), //接口类型
new WSHttpBinding(), //传输协议
"CalculatorService"); //服务名称
```

必须指定终结点公开的协议、绑定和终结点的地址。对于此示例，将 `IService` 指定为协定，将 `WSHttpBinding` 指定为绑定，并将 `CalculatorService` 指定为地址。

 **注意：**此处的 `CalculatorService` 是相对地址，完整地址是基址和终结点地址的组合。

(3) 启动元数据交换，代码如下：

```
//激活元数据交换
ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
smb.HttpGetEnabled = true; //启用HttpGet
selfHost.Description.Behaviors.Add(smb); //添加元数据
```

(4) 启动和终止 WCF 服务，代码如下：

```
//启动服务。
selfHost.Open();
//在控制台显示运行信息
Console.WriteLine("服务已经启动.");
Console.WriteLine("按 <ENTER> 终止服务.");
Console.WriteLine();
Console.ReadLine();
//终止服务
selfHost.Close();
```

(5) 添加启动服务和终止服务的异常处理信息，即 `try-catch` 处理，完整的启动服务代码如下所示。

```
static void Main(string[] args)
{
    //创建基本地址
    Uri baseAddress = new Uri("http://localhost:8000/ServiceModel
Samples/Service");
    //创建服务主机
    ServiceHost selfHost = new ServiceHost(typeof(CalculatorService),
baseAddress);
    try
    {
        //创建服务终结点
        selfHost.AddServiceEndpoint(
            typeof(IService), //接口类型
            new WSHttpBinding(), //传输方式
            "CalculatorService"); //接口服务
        //激活元数据交换
        ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
        smb.HttpGetEnabled = true; //启用HttpGet
        selfHost.Description.Behaviors.Add(smb); //添加元数据
        //启动服务。
        selfHost.Open();
        //在控制台显示信息
        Console.WriteLine("服务已经启动.");
        Console.WriteLine("按 <ENTER> 终止服务.");
        Console.WriteLine();
    }
}
```



```

        Console.ReadLine();
        // 终止服务
        selfHost.Close();
    }
    catch (CommunicationException ce)
    {
        Console.WriteLine("异常: {0}", ce.Message);
        //调用终止服务方法
        selfHost.Abort();
    }
}

```

运行该服务，结果如图 28.4 所示。



图 28.4 启动 WCF 服务

此时如果按 Enter 键，则将会终止该服务。为了后续章节中 WCF 客户端的成功创建和调用，这里需要保持服务端的启动状态。

28.2.5 创建 WCF 客户端程序

WCF 客户端是可以调用 WCF 服务的任意客户端的统称。在调用 WCF 之前，需要首先生成 WCF 的代理配置文件，通过代理文件的配置，以及在客户端本地生成的代理对象，来实现 WCF 客户端对服务端的访问。本节首先介绍如何创建客户端程序。

(1) 创建 WCF 客户端程序。选择菜单栏的“文件”|“新建”|“项目”命令，在弹出的“新建项目”对话框中新建一个“控制台应用程序”的项目，项目名称命名为 WCFClient。

(2) 添加组件。在项目的“解决方案资源管理器”对话框中右键单击项目名称 WCFService，在弹出的菜单中单击“添加引用”菜单在弹出的“添加引用”对话框中单击“.NET”选项卡，然后确保选中“组件名称”为 System.ServiceModel 的组件之后，单击“确定”按钮完成组件的引用，如图 28.3 所示。

(3) 添加命名空间。双击项目中的 Program.cs 文件，在代码编辑区域的命名空间引用代码区域，添加新的命名空间的引用代码，如下所示。

```
using System.ServiceModel;
```

28.2.6 配置 WCF 客户端的配置

客户端的配置文件描述，可以通过工具生成，也可以通过 Visual Studio 2010 向导完成

创建，本节将介绍如何通过 Visual Studio 2010 创建 WCF 客户端的配置。

(1) 添加服务代理。确保 WCF 服务端正常运行的状态下，右键单击项目 WCFClient，在弹出的菜单中选择“添加服务引用”命令，在弹出的“添加服务引用”对话框的“地址”栏中输入 WCF 服务端的地址信息为 `http://localhost:8000/ServiceModelSamples/Service`，然后单击“前往”按钮，待 Visual Studio 2010 进行一段时间的搜索之后，会在“服务”区域显示前面创建完毕的服务名称，以及对应的接口名称和接口对应的方法。

 注意：接口方法名称是在“操作”区域内显示的。

最终将会发现服务的“添加服务引用”窗口，如图 28.5 所示。



图 28.5 添加 WCF 服务引用

(2) 生成代理客户端配置文件。单击“确定”按钮，在该 WCF 客户端项目中，会自动根据 WCF 服务添加一个名为 `app.config` 的配置文件。该文件的内容如下：

```
<!--定义配置文件版本以及编码方式-->
<?xml version="1.0" encoding="utf-8" ?>
<!--配置文件节点-->
<configuration>
  <!--定义服务模型节点-->
  <system.serviceModel>
    <!--定义绑定信息-->
    <bindings>
      <!--wsHttp 绑定配置部分-->
      <wsHttpBinding>
        <!--定义绑定名称,关闭超时时限, 打开超时时限,接收超时时限,发送超时时限-->
        <binding name="WSHttpBinding_IService" closeTimeout="00:01:00"
          openTimeout="00:01:00" receiveTimeout="00:10:00"
          sendTimeout="00:01:00"
          <!--定义是否经过本地代理,交易流类型,主机名称认证模式-->
          bypassProxyOnLocal="false" transactionFlow="false" host
            NameComparisonMode="StrongWildcard"
          <!--最大缓冲池字节数,最大接收消息字节数-->
```



```

maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
<!--定义消息编码,文本编码,是否使用默认代理,是否使用 cookie-->
messageEncoding="Text" textEncoding="utf-8" useDefault
WebProxy="true" allowCookies="false">
<!--定义终结点处理的 SOAP 消息的复杂性约束-->
<readerQuotas maxDepth="32"
  <!--定义最大字符串内容长度,最大数组长度-->
  maxStringContentLength="8192" maxArrayLength="16384"
  <!--定义每次读取最大返回字节数,表名称允许的最大字符数-->
  maxBytesPerRead="4096" maxNameTableCharCount="16384" />
<!--定义服务可靠会话绑定元素属性的便捷访问。消息传递与消息发送顺序
是否保持一致,服务在关闭之前保持非活动状态的时间间隔,默认启用状态-->
<reliableSession ordered="true" inactivityTimeout="00:10:00"
  enabled="false" />
<!--定义安全模式-->
<security mode="Message">
  <!--定义传输客户端的身份认证模式,代理认证模式-->
  <transport clientCredentialType="Windows"
    proxyCredentialType="None" realm="" />
  <!--定义消息客户端的身份认证模式,代理认证模式-->
  <message clientCredentialType="Windows"
    <!--定义指定是否已进行消息级 SSL 协商以获取服务的证书-->
    negotiateServiceCredential="true"
    <!--定义指定要与 BasicHttpMessageSecurity 一起使用的算法组-->
    algorithmSuite="Default"
    <!--定义获取或设置布尔值,该值指定是否建立安全上下文令牌-->
    establishSecurityContext="true" />
  </message>
</transport>
</security>
</binding>
</wsHttpBinding>
</bindings>
<!--定义客户端属性-->
<client>
  <!--定义终结点属性,包括终结点地址,绑定方式,绑定配置,契约和接口名称-->
  <endpoint address="http://localhost:8000/ServiceModelSamples/
    Service/CalculatorService"
    binding="wsHttpBinding"
    bindingConfiguration="WSHttpBinding_IService"
    contract="ServiceReference.IService"
    name="WSHttpBinding_IService">
  <!--定义认证信息-->
  <identity>
    <userPrincipalName value="Demo\Administrator" />
    <!--定义默认用户-->
  </identity>
</endpoint>
</client>
</system.serviceModel>
</configuration>

```

28.2.7 WCF 客户端对服务端的调用

在实现了 WCF 客户端的配置后,就可以完成 WCF 客户端对服务端的调用功能。

(1) 创建客户端终结点。首先定义访问服务端的客户端终结点代码,如下所示。

```
//创建客户端终结点
```



```
EndpointAddress epAddress = new EndpointAddress(
"http://localhost:8000/ServiceModelSamples/Service /CalculatorService");
ServiceReference.ServiceClient client = new ServiceReference.ServiceClient
(new WSHttpBinding(), epAddress);
```

(2) 实现 WCF 服务端代码调用。直接使用生成的本地代理对象,实现对 WCF 服务端方法的调用,代码如下:

```
//调用 WCF 服务端加方法
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
//显示加方法调用结果
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//调用 WCF 服务端减方法.
value1 = 145.00D;
value2 = 76.54D;
result = client.Subtract(value1, value2);
//显示减方法调用结果
Console.WriteLine("Subtract({0},{1}) = {2}",
    value1, value2, result);

//调用 WCF 服务端乘方法
value1 = 9.00D;
value2 = 81.25D;
result = client.Multiply(value1, value2);
//显示乘方法调用结果
Console.WriteLine("Multiply({0},{1}) = {2}",
    value1, value2, result);

//调用 WCF 服务端除方法
value1 = 22.00D;
value2 = 7.00D;
result = client.Divide(value1, value2);
//显示除方法调用结果
Console.WriteLine("Divide({0},{1}) = {2}",
    value1, value2, result);

//关闭客户端
client.Close();
//显示提示信息
Console.WriteLine();
Console.WriteLine("按 <ENTER> 终止客户端.");
Console.ReadLine();
```

(3) 运行该 WCF 客户端程序,运行结果如图 28.6 所示。



图 28.6 运行 WCF 客户端程序

此时对应的服务端控制台窗口，会输出客户端的调用信息，如图 28.7 所示。



图 28.7 运行 WCF 客户端程序的服务端结果

28.3 主要的 WCF 技术

WCF 除了基本的服务以及客户端调用技术之外，还包括许多在性能、服务约定、安全等方面的技术特点，本节将介绍几个主要的 WCF 技术。

28.3.1 使用会话在客户端与服务间交互

WCF 会话是指在客户端与服务之间进行的消息交互的一种模式，例如一个服务中的对象实例在每次与客户端交互中都存在，还是每次与客户端交互时重新创建该对象的实例等。

注意：在 WCF 对话中，涉及服务的实例化概念，实例化是指用户定义的服务对象，以及相关对象的生存期控制。

WCF 对话是通过 `System.ServiceModel.ServiceContractAttribute.SessionMode` 属性设置在服务协定中的，该属性是一个枚举值，包括 3 种状态，分别是 `Allowed`、`Required` 和 `NotAllowed`。`Allowed` 指定当传入绑定支持会话时，协定也支持会话；`Required` 指定协定需要会话绑定，如果绑定并未配置为支持会话，则将引发异常；`NotAllowed` 指定协定永不支持启动会话的绑定。例如 28.2.2 节中的服务协定可以定义为允许状态的会话属性，代码如下：

```
[ServiceContract(Namespace = "http://VS2010.WCF.Sample",
    SessionMode= SessionMode.Required)]
public interface IService
```

实例化控制如何创建 `InstanceContext` 以响应传入的消息。在默认情况下，每个 `InstanceContext` 都与一个用户定义服务对象相关联，因此设置 `InstanceContextMode` 属性也可以控制用户定义服务对象的实例化。`InstanceContextMode` 枚举定义了实例化模式。`InstanceContextMode` 定义了 3 种枚举值，分别是 `PerCall`、`PerSession` 和 `Single`。`PerCall` 为每个客户端请求创建一个新的 `InstanceContext`；`PerSession` 为每个新的客户端会话创建一个

新的 `InstanceContext`，并在该会话的生存期内对其进行维护；`Single` 单个 `InstanceContext` 处理应用程序生存期内的所有客户端请求。设定实例化属性的服务协定的代码如下：

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
public interface IService
```

在 WCF 中，会话、实例化的模式，以及会话通道的行为之间有着互相制约的关系，在不同的情况下，对服务协定的要求也不同。具体的要求如表 28.1 所示。

表 28.1 会话、实例化、会话通道行为关系

InstanceContextMode值	Required	Allowed	NotAllowed
PerCall	会话通道的行为：每个调用都具有一个会话和 <code>InstanceContext</code> 无会话通道的行为：将引发异常	会话通道的行为：每个调用都具有一个会话和 <code>InstanceContext</code> 无会话通道的行为：每个调用都具有一个 <code>InstanceContext</code>	会话通道的行为：将引发异常 无会话通道的行为：每个调用都具有一个 <code>InstanceContext</code>
PerSession	会话通道的行为：每个通道都具有一个会话和 <code>InstanceContext</code> 无会话通道的行为：将引发异常	会话通道的行为：每个通道都具有一个会话和 <code>InstanceContext</code> 无会话通道的行为：每个调用都具有一个 <code>InstanceContext</code>	会话通道的行为：将引发异常 无会话通道的行为：每个调用都具有一个 <code>InstanceContext</code>
Single	会话通道的行为：所有调用都具有一个会话和一个 <code>InstanceContext</code> 无会话通道的行为：将引发异常	会话通道的行为：所创建的单一实例或用户指定的单一实例都具有一个会话和 <code>InstanceContext</code> 无会话通道的行为：所创建的单一实例或用户指定的单一实例都具有一个 <code>InstanceContext</code>	会话通道的行为：将引发异常 无会话通道的行为：所创建的每个单一实例或用户指定的单一实例都具有一个 <code>InstanceContext</code>

28.3.2 WCF 事务管理模型

事务是指提供一种分组方法，将一组操作分为单个不可分的执行单元。它具有原子性、一致性、隔离性和持续性的特点。

WCF 提供了丰富的事务管理模型，可以通过在客户端开启并使用事务，也可以通过服务端设定来控制事务的各个类型。可以通过下面的方式实现 WCF 中对事务的控制功能。

- ☐ 使用 `ServiceBehaviorAttribute` 属性配置事务超时值，以及隔离级别的筛选。
- ☐ 启用事务功能并使用 `OperationBehaviorAttribute` 属性配置事务完成行为。
- ☐ 使用协定方法上的 `ServiceContractAttribute` 和 `OperationContractAttribute` 属性来要求、允许或拒绝事务流。

`ServiceBehaviorAttribute` 属性指定服务协定实现的内部执行行为。它包括以下几点。

- ☐ `TransactionAutoCompleteOnSessionClose`：此属性指定会话关闭时是否完成未完成的事务。
- ☐ `ReleaseServiceInstanceOnTransactionComplete`：此属性指定事务完成时是否释放基

础服务实例。

- ❑ **TransactionIsolationLevel**: 此属性指定用于服务内事务的隔离级别。
- ❑ **TransactionTimeout**: 此属性指定一个时间段, 在服务中创建的新事务必须在此时间段内完成。

OperationBehaviorAttribute 属性指定服务实现中方法的行为, 它包括以下两点。

- ❑ **TransactionScopeRequired**: 指定是否必须在活动事务范围内执行方法。
- ❑ **TransactionAutoComplete**: 指定在没有引发未处理的异常情况下, 在其中执行方法的事务是否自动完成。

例如 28.2.2 节中的服务协定如果定义为使用事务的代码, 则如下所示。

```
[ServiceContract(Namespace = "http://VS2010.WCF.Sample")]
public interface IService
{
    //加操作方法
    [OperationContract]
    [OperationBehavior(TransactionAutoComplete = true,
        TransactionScope Required = true)]
    double Add(double n1, double n2);
    //减操作方法
    [OperationContract]
    [OperationBehavior(TransactionAutoComplete = true,
        TransactionScope Required = true)]
    double Subtract(double n1, double n2);
    //乘操作方法
    [OperationContract]
    [OperationBehavior(TransactionAutoComplete = true,
        TransactionScope Required = true)]
    double Multiply(double n1, double n2);
    //除操作方法
    [OperationContract]
    [OperationBehavior(TransactionAutoComplete = true,
        TransactionScope Required = true)]
    double Divide(double n1, double n2);
}

//服务行为属性
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single,
    InstanceContextMode = InstanceContextMode.PerSession,
    ReleaseServiceInstanceOnTransactionComplete = true,
    TransactionIsolationLevel =
        System.Transactions.IsolationLevel.ReadCommitted,
    TransactionTimeout="60")]
public class CalculatorService : IService
```

28.4 本章总结


本章主要介绍了 WCF 框架, 包括 WCF 的基本架构、组成模型, 以及如何通过 Visual Studio 2010 创建 WCF 应用程序。通过本章的学习, 使读者更加深刻地了解了服务协定、服务、客户端、终结点等 WCF 中的基本概念。

28.5 实战练习

本章练习中要求创建一个 WCF 服务程序，用来模拟自动应答机器人。根据客户端的调用，该服务程序能随机返回一个字符串作为应答。

1. 在 Visual Studio 2010 中新建一个控制台应用程序 MyWCFService，将此应用程序创建为 WCF 服务程序，编写代码定义一个接口 IAnswer，在接口中定义一个响应用户的方法。

2. 接第 1 题，在 MyWCFService 中实现 IAnswer 接口中的方法，并在 Main 方法中实现 WCF 服务。

 **提示：**可定义一个数组保存机器人的答案，当客户调用 WCF 服务时，随机给出一个数组中的字符串即可。


3. 在 Visual Studio 2010 中新建一个控制台应用程序 MyWCFClient，作为 WCF 的客户端程序，编写代码调用 MyWCFService 中编写的 WCF 服务。

第 29 章 Windows WF 框架

Windows Workflow Foundation (WF) 是微软提供的一个统一的工作流平台（在 .NET 3 中缩写为 WWF，由于与 World Wild Fund for Nature “世界自然基金会”的缩写相同，因此微软将其改称为 WF）。在 WF 平台中，可以设计出各种基于不同应用，以及基于不同服务载体工作流应用。WF 是一个功能强大的工作流框架，可以让用户在其基础上，开发出适合 Windows Vista、Windows XP、Windows 7，以及 Windows Server 2003/2008 等操作系统的人工工作流。

29.1 C#的工作流开发框架

WF 作为一个工作流的开发框架，提供了强大的流程设计、流程引擎、流程活动、流程服务等元素。本章将介绍 WF 的技术框架，以及各个元素在 WF 中的作用。

说明：工作流开发在软件行业中，已经经历了很长的时间了，市场上的工作流产品也多种多样，不能说 WF 是工作流的产品，而应该属于框架的范畴。

29.1.1 了解 WF 框架

WF 提供了与其他 .NET Framework 4 技术（如 WCF 和 WPF）一致和熟悉的开发体验。WF API 完全支持 VB .NET 和 C#、专用工作流编译器、在工作流中调试、图形工作流设计器，并支持完全用代码或标记开发工作流。WF 还提供了可扩展模型和设计器，用于生成成为最终用户或跨多个项目重用封装工作流功能的自定义活动。WF 的框架图如图 29.1 所示。

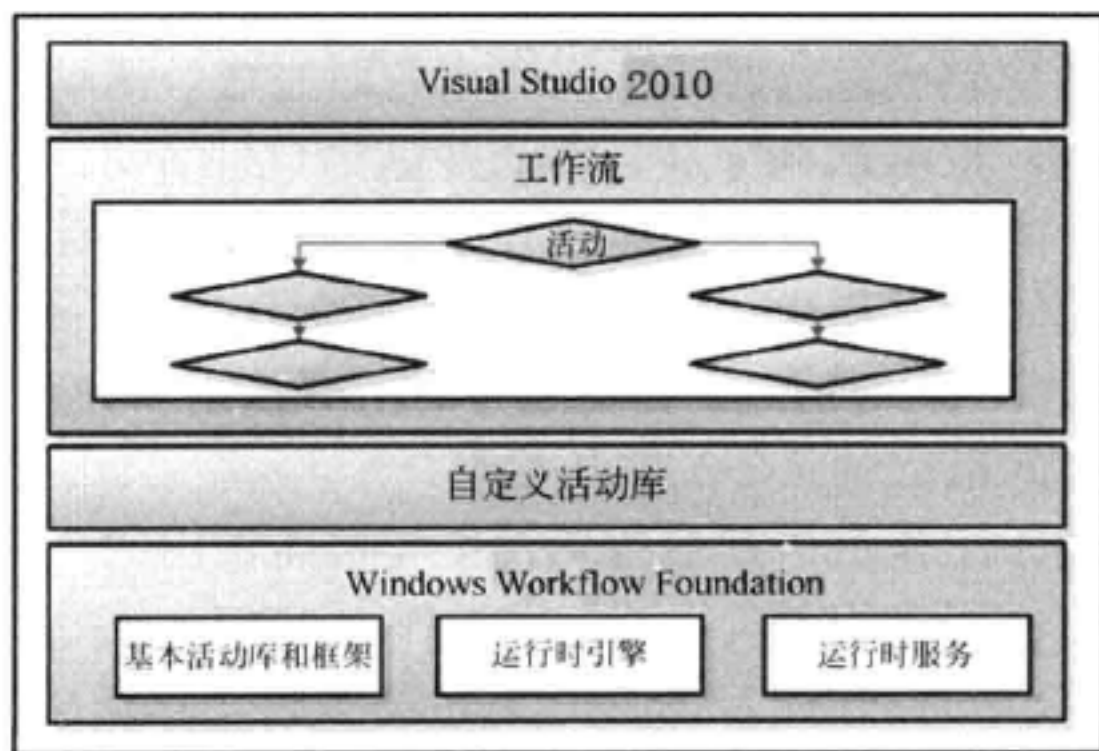



图 29.1 WF 框架图

从图中可以看出, WF 分为开发工具、工作流、自定义活动库, 以及 WF 运行时等组成元素。而工作流、自定义活动库, 以及运行时 3 个元素又运行在 WF 的主机进程中, 由主机进程统一进行管理。

前面已经介绍过 WF 应用可以通过 Visual Studio 2010 进行方便的图形开发或者以程序的方式进行代码的定制开发。通过 Visual Studio 2010 可以创建工作流应用、自定义活动, 以及自定义服务库等基本组件。经过 Visual Studio 2010 定制开发之后, 就形成了工作流。这些工作流由多个活动组成, 各个活动之间通过逻辑或者数据关系进行流程联系, 从而实现完整的流程运转。在工作流中, 可以使用 WF 本身提供的活动库, 也可以自定义开发自定义的活动库。这些活动库可以将通用的流程处理应用进行定制开发, 以便于在新的工作流应用中可以重复使用, 提高开发的效率。在 WF 的最底层是 WF 核心部分, 即 WF 的运行时组件。在该组件中, 包括流程中使用的基本活动库以及流程框架, 有工作流进行流转的运行时引擎, 以及工作流服务所需要的载体运行时服务等基本组件。

 **说明:** 每个正在运行的工作流实例都是由进程中运行时引擎创建和维护的, 该引擎通常称为工作流运行时引擎。在一个应用程序域中可以有多个工作流运行时引擎, 并且运行时引擎的每个实例均可支持多个并发运行的工作流实例。

工作流模型经过编译后, 可以在包括控制台应用程序、基于窗体的应用程序、Windows 服务、ASP.NET 网站和 Web 服务在内的任意 Windows 进程中执行。由于工作流是在进程中承载, 因此工作流可以轻松地与其宿主应用程序通信。

29.1.2 WF 框架中的重要元素

在 WF 框架中, 包括很多的框架组成元素, 这些元素有的是设计阶段的组成部分, 有的是运行时阶段的包含内容。但无论是设计阶段还是运行时阶段, 它们都为整个流程框架提供了基础的服务。下面对框架中涉及的重要元素进行逐一说明。

- ❑ **工作流:** 工作流是将各个操作组合在一起, 形成了有一定特点或者规则的流程集合。工作流提供了一种方法, 用于描述多项短期运行或长期运行的工作之间的执行顺序和依赖关系。此工作从头到尾地贯穿模型, 并且活动可以人工执行或由系统功能执行。
- ❑ **活动:** 活动是工作流的基本单元。以编程方式将活动添加到工作流中, 当定流路径中的所有活动都完成运行时, 工作流实例即完成。活动可以执行单个操作, 如向数据库写入值, 也可以执行复合活动并包含一组活动。活动有两种行为类型, 即运行时和设计时。运行时行为在执行时指定操作。设计时行为在设计器中显示时可以控制活动的外观及其交互。
- ❑ **服务:** 当工作流实例运行时, 工作流运行时引擎使用多种服务。WF 提供可满足多种应用程序需要的运行时服务的默认实现。
- ❑ **补偿:** 补偿是由于工作流中其他位置发生异常而做出的一种行为, 这种行为撤销由成功完成的可补偿活动所执行的任何操作。
- ❑ **持久性:** WF 简化了有状态的、长期运行的持久性工作流应用程序的创建过程。工作流运行时引擎管理工作流的执行情况, 而且允许工作流长期保持活动状态并在应用程序重新启动之后存在。

- 跟踪：跟踪是一项功能，用于指定并捕获有关 workflow 实例的信息，并在这些实例执行时存储该信息，便于以后对这些存储信息进行统计分析。
- 序列化：对 workflow、活动和规则可以进行序列化和反序列化。这样就可以保持它们，以及在工作流标记文件中使用它们，在工作流设计器中查看其属性、字段和事件。
- 错误处理：workflow 在运行到某个活动时，可能由于设计原因或者运行时数据的不完整等原因，从而导致流程发生了可以恢复或者不可恢复的错误，为了使流程可以继续运转，需要对这些没有想到的异常和错误进行统一处理，以使最终用户或者跟踪文件可以发现错误，提供错误的解决方法。
- 工作流标记：基于可扩展应用程序标记语言 XAML，工作流标记可以使开发人员和设计人员以声明方式为业务逻辑建模，并将其与代码文件进行区分。workflow 可以以声明方式建模，所以可以在运行时，通过直接将工作流标记文件加载到 workflow 运行时引擎的方式来激活 workflow。

29.2 开发 WF 工作流应用程序

WF 应用开发可以通过完全的代码编写实现，也可以通过集成开发工具 Visual Studio 2010 来完成开发。本章将介绍如何使用 Visual Studio 2010 开发 WF 工作流应用程序，以及在开发工作流程序中需要使用到的活动和服务的基本概念。


29.2.1 第一个 WF 应用程序

WF 应用程序可以通过统一的开发平台 Visual Studio 2010 进行创建，在 Visual Studio 2010 中提供了多种 WF 相关的项目类型，包括顺序工作流、状态机工作流等。

例如，要创建这样一个工作流：在控制台接收用户输入，如果输入内容为 Test，则在控制台输出 Hello Test!，如果输入的内容不为 Test，则在控制台输出 Hello world。

对于这个例子的要求，通过 C# 代码编写程序可以很容易实现，但现在要求用 WF 来创建其流程，具体操作步骤如下所述。

(1) 创建 WF 类型的项目。首先启动 Visual Studio 2010，在菜单栏中单击“文件”|“新建”|“项目”菜单，在弹出的“新建项目”对话框中，展开左侧“已安装的模板”区域中的 Visual C# 节点，然后单击 Workflow 菜单，在右侧的“模板”区域将列出 WF 相关的项目类型。

 注意：也可以通过其他的 .NET 语言进行 WF 的项目开发，如果使用 VB.NET 开发，则在左侧的“已安装的模板”区域选择相应的语言节点，然后单击 Workflow 菜单进行选择。

通过 Visual Studio 2010 可以创建 4 种 WF 项目类型，分别是活动设计器库、活动库、WCF 工作流服务应用程序、工作流控制台应用程序，如图 29.2 所示。



图 29.2 创建 WWF 项目

在图 29.2 所示的对话框中，单击“确定”按钮，此时在 Visual Studio 2010 中会自动创建一个工作流项目。

(2) 查看流程设计图。在新建的项目中，Visual Studio 2010 会自动在项目中添加一个 Workflow1.xaml 文件，并显示该 XAML 文件的图形界面，开发人员可以通过从“工具箱”中拖曳相关的活动控件到设计图中，来完成流程的图形化设计过程。

注意：如果“工具箱”窗口没有打开，可以通过选择菜单栏中的“视图”|“工具箱”命令将其打开。

(3) 从“工具箱”的“流程图”组中拖动 Flowchart 到设计图中（有关各种内置活动的功能和作用在下一节中介绍，这里只需要按步骤进行操作即可）。

(4) 在设计界面左下角单击“变量”选项，打开如图 29.3 所示的变量列表，输入一个名为 UserName 的变量。



图 29.3 添加变量

(5) 从“工具箱”的“基元”组中拖动 InvokeMethod 活动到 Flowchart 中。

(6) 将鼠标指针移到 Start 图标，在该图标四周将显示四个小方块，拖动方块到刚拖入的 InvokeMethod 框上，创建一个流程线。

(7) 在 InvokeMethod 活动中单击 TargetType 右侧的下拉框，从中选择“浏览类型”

命令，将打开如图 29.4 所示对话框。在上方的文本框中输入 System.Console，在输入的过程中，下方将逐步缩小范围，可看到 Console。单击“确定”按钮即可返回 InvokeMethod 活动。



图 29.4 浏览并选择.NET 类型

(8) 在 InvokeMethod 活动的 MethodName 右侧的文本框中输入 ReadLine。这两步的操作就是在流程中引用一个 System.Console.ReadLine 方法，用来接收用户输入的内容。那么，用户输入的内容保存在哪里呢？在第 (4) 步中设置了变量 UserName，显然要将输入的内容保存在该变量中。在设计视图中单击 InvokeMethod，然后打开“属性”窗口，在 Result 文本框中输入 UserName。经过以上几步设计的界面如图 29.5 所示。



图 29.5 设计视图

(9) 接下来创建流程中的判断。用类似的方法从“工具箱”的“控制流”中拖动 If 到设计图中，再双击 If 框打开 If 的设计界面，在 Then 和 Else 中各拖入“基元”组中的一个 WriteLine。然后设置 If 的条件 Condition 中的判断条件，这里输入“UserName = "Test"”（不是 C#中的相等判断符“==”，而是 VB.NET 中的语法格式），接着在 WriteLine 中设置输出的文本如图 29.6 所示。



图 29.6 If 活动设计视图

(10) 如图 29.6 所示，在设计图左上角有一个活动导航，单击 Flowchart 可退出 If 的设计界面，显示如图 29.7 所示设计界面。



图 29.7 设计视图

(11) 至此，要求的流程已经创建完成，接下来就可以运行看看效果。按 F5 键启动程序，将打开控制台，由于工作流此时等待用户输入，因此一直显示黑屏。输入 Test，按 Enter 键，将看到如图 29.8 左图所示的结果。再次启动程序，输入一个 abc（不为 Test 即可），按 Enter 键，将看到如图 29.8 右图所示结果。



图 29.8 工作流运行结果

从上面的操作过程可看到,在创建工作流时,并没有编写一行代码,就可完成条件的判断。对于熟悉 C# 的你来说,会感觉对于这种工作流直接编写 C# 代码要快得多。不过,对于一些工作流复杂,且工作流设计人员与代码编写人员分开的情况下,使用工作流就具有了快速、直观、可重用等很多优点。

对于工作流设计人员设计制作的工作流以 XAML 格式保存,这种图形方式的工作流直观、修改方便。对于上例中创建的工作流,其 XAML 文件如下所示。

```
<Activity mc:Ignorable="sap" x:Class="DemoWFProject.Workflow1"
    <!--此处省略多行名称空定义-->
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!--定义 Flowchart -->
    <Flowchart sad:XamlDebuggerXmlReader.FileName=
        "\DemoWFProject\Workflow1.xaml"
        sap:VirtualizedContainerService.HintSize="656,636">
    <!--定义变量 -->
    <Flowchart.Variables>
        <Variable x:TypeArguments="x:String" Name="UserName">
            <Variable.Default>
                <Literal x:TypeArguments="x:String" Value="" />
            </Variable.Default>
        </Variable>
    </Flowchart.Variables>
    <!--管理状态 -->
    <sap:WorkflowViewStateService.ViewState>
        <scg3:Dictionary x:TypeArguments="x:String, x:Object">
            <x:Boolean x:Key="IsExpanded">False</x:Boolean>
            <av:Point x:Key="ShapeLocation">270,2.5</av:Point>
            <av:Size x:Key="ShapeSize">60,75</av:Size>
            <av:PointCollection x:Key="ConnectorLocation">
                300,77.5 300,92.5</av:PointCollection>
            <x:Double x:Key="Width">642.5</x:Double>
        </scg3:Dictionary>
    </sap:WorkflowViewStateService.ViewState>
    <!--定义 startNode -->
    <Flowchart.StartNode>
        <FlowStep x:Name="__ReferenceID1">
            <sap:WorkflowViewStateService.ViewState>
                <scg3:Dictionary x:TypeArguments="x:String, x:Object">
                    <av:Point x:Key="ShapeLocation">187.5,92.5</av:Point>
                    <av:Size x:Key="ShapeSize">225,135</av:Size>
                    <av:PointCollection x:Key="ConnectorLocation">
                        300,227.5 300,243.5</av:PointCollection>
                </scg3:Dictionary>
            </sap:WorkflowViewStateService.ViewState>
            <!--定义 InvokeMethod -->
            <InvokeMethod sap:VirtualizedContainerService.HintSize="225,135"
                MethodName="ReadLine" TargetType="s:Console">
                <InvokeMethod.Result>
                    <OutArgument x:TypeArguments="x:String">
                        [UserName]</OutArgument>
                    </InvokeMethod.Result>
                </InvokeMethod>
            <FlowStep.Next>
                <FlowStep x:Name="__ReferenceID0">
                    <sap:WorkflowViewStateService.ViewState>
                        <scg3:Dictionary x:TypeArguments="x:String, x:Object">
                            <av:Point x:Key="ShapeLocation">200,243.5</av:Point>
```

```

        <av:Size x:Key="ShapeSize">200,53</av:Size>
        <av:PointCollection x:Key="ConnectorLocation">
            300,296.5 300,326.5 300,332.5</av:PointCollection>
        </scg3:Dictionary>
    </sap:WorkflowViewStateService.ViewState>
    <!--条件判断 -->
    <If Condition="[UserName = &quot;Test&quot;]"
        sap:VirtualizedContainerService.HintSize="200,53">
        <If.Then>
            <WriteLine sap:VirtualizedContainerService.HintSize=
                "219,100" Text="Hello Test!" />
        </If.Then>
        <If.Else>
            <WriteLine sap:VirtualizedContainerService.HintSize=
                "220,100" Text="Hello World" />
        </If.Else>
        </If>
    </FlowStep>
    </FlowStep.Next>
</FlowStep>
</Flowchart.StartNode>
<x:Reference>__ReferenceID0</x:Reference>
<x:Reference>__ReferenceID1</x:Reference>
</Flowchart>
</Activity>

```

对于以上 XAML 代码创建的工作流,怎样才能运行呢?这就需要在程序中通过 `WorkflowInvoker` 对象的 `Invoke` 方法来调用设计好的工作程序。因此,在 `Program.cs` 文件中可看到如下代码:

```

class Program
{
    static void Main(string[] args)
    {
        WorkflowInvoker.Invoke(new Workflow1());
        Console.ReadLine();
    }
}

```

在以上代码中, `Invoke` 方法的参数为用 `new` 关键字创建的工作流对象。使用 `Invoke` 的重载方法还可以向调用的工作流传入参数。

29.2.2 WF 工作流的基本元素: WF 活动

由上面的例子可看出,WF 活动是组成 WF 工作流的基本元素,可以通过活动完成流程的逻辑控制、判断、循环、调用等基本操作。在工作流中,可以使用 WF 本身提供的活动,也可以使用开发人员自己定制开发的自定义活动。

在 Visual Studio 2010 的“工具箱”中可看到内置了很多可用的活动,通过这些活动可非常方便地对业务流程建模(与之前版本相比,内置活动有较大变化,如果用过 .NET 3 或 .NET 3.5 的 WF 框架,这里需注意了),下面简单介绍系统内置的活动。

从工具箱中可看到, .NET 4 将活动控件进行了分类,下面分别介绍各分类中的活控件的作用。

1. 控制流

控制流组中的活动控件用来在较大工作流中表示循环和分支等情况,类似于 C#中的相应代码,如图 29.9 所示。



图 29.9 控制流

- ❑ **DoWhile 活动:** 执行其 Body 中包含的活动至少一次,直到指定条件的计算结果为 false。如果需要执行循环体中包含的活动零次或多次,可用 While 活动。
- ❑ **ForEach<T>活动:** 对于指定 Values 集合中的每一项,ForEach 活动执行包含在它的 Body 中的活动。
- ❑ **If 活动:** 计算条件并根据该计算结果执行活动。当使用编程的过程建模样式时,此活动最有用。例如,If 活动可嵌套在 Sequence 活动或 Parallel 活动内。
- ❑ **Parallel 活动:** 并发执行一组子活动。
- ❑ **ParallelForEach<T>活动:** 枚举集合元素并对集合中的每个元素并行执行嵌入语句,这将在同一线程上异步执行。
- ❑ **Pick 活动:** 提供基于事件的控制流。该活动执行多个分支中的一个分支来响应某个触发的事件。
- ❑ **PickBranch 活动:** PickBranch 在可由传入事件触发的 Pick 活动中提供基于事件的执行路径。
- ❑ **Sequence 活动:** 该活动包含子活动的已排序集合,将按该排序执行这些子活动。
- ❑ **Switch<T>活动:** Switch 活动计算指定表达式并执行活动集合中其关联键与计算所得值匹配的活动。Switch<T>活动设计器用于在 Windows 工作流设计器中创建和配置 Switch 活动。
- ❑ **While 活动:** 当指定的 Condition 的计算结果为 true 时,While 活动执行其 Body 中包含的活动。包含的活动可能永远都不会执行。

2. 流程图

流程图工作流是 .NET 4 中的全新概念。流程图活动非常重要,常常是放置在 WF 设计

器上的第一项。如图 29.10 所示是内置的流程图活动，可以使用这些流程图活动来构建工作流。



图 29.10 流程图

- ❑ **Flowchart 活动：**用于创建定义和管理复杂流控制的工作流。可以使用代码或工作流设计器创作 Flowchart。Flowchart 指定工作流启动时执行的唯一 StartNode，并使用链接的 Nodes 网络创建任意循环或将执行流在任意给定时间转移到工作流中的其他任何位置。
- ❑ **FlowDecision 活动：**FlowDecision 节点是一个条件节点，它根据指定条件是否成立来将控制流分支到两个备选分支之一。
- ❑ **FlowSwitch<T>活动：**是一个条件节点，它在需要两个以上备选分支时根据匹配条件分支控制流。如果流分支仅需要两个路径，可改用 FlowDecision 活动。

3. 消息传递

工作流可以使用消息传递活动轻松地调用外部 XML Web 服务或 WCF 服务的成员，且可以接收外部服务的通知。如图 29.11 所示是内置消息传递活动。



图 29.11 消息传递

- ❑ **CorrelationScope 活动：**该活动使用 CorrelationHandle 对象提供子消息传递活动的隐式管理。
- ❑ **InitializeCorrelation 活动：**该活动用于在发送或接收消息之前在这些消息之间建立相关。

- ❑ **Receive 活动**: 这是接收消息的活动, 可接收的消息包括内置类型(如 `Message`、`Stream` 或 `XElement`) 或者应用程序定义的数据协定、消息协定或可序列化的 XML 类。
- ❑ **ReceiveAndSendReply 活动**: 用于在 `Sequence` 活动中创建一对预配置的 `Receive` 和 `SendReply` 活动, 这对活动作为服务器上请求/响应消息交换模式的一部分而关联。
- ❑ **Send 活动**: 用于向服务发送消息。作为客户端上请求/响应消息交换模式的一部分接收消息的 `ReceiveReply` 活动可绑定到 `Send` 活动。
- ❑ **SendAndReceiveReply 活动**: 用于在 `Sequence` 活动中创建一对预配置的 `Send` 和 `ReceiveReply` 活动, 这对活动作为客户端上请求/响应消息交换模式的一部分而关联。
- ❑ **TransactedReceiveScope 活动**: 能够将事务流动到工作流或调度程序创建的服务器事务中。

4. 运行时

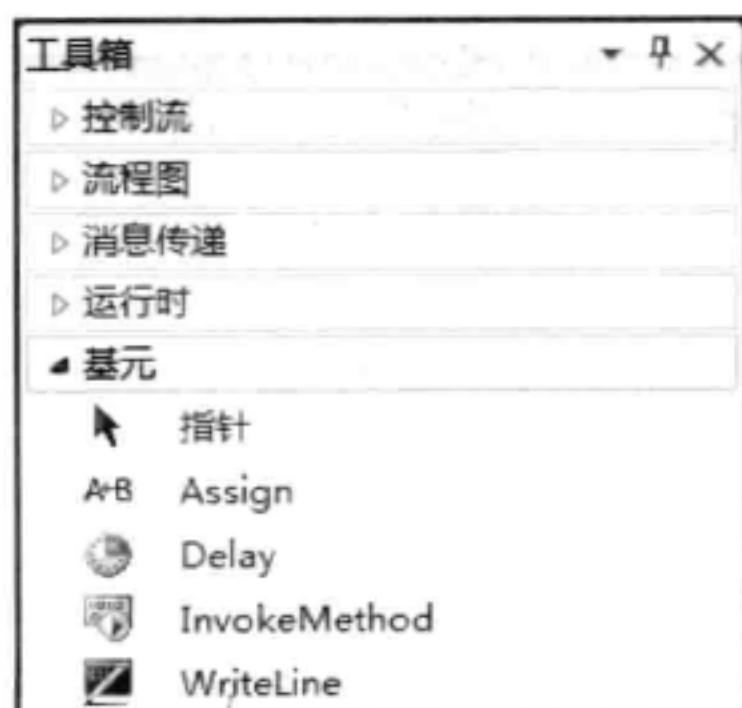
运行时活动可用来构建调用了工作流运行时的工作流, 如图 29.12 所示是运行时活动。



- ❑ **Persist 活动**: 用于将工作流保存到磁盘中(如有可能)。`Persist` 活动无法在非持久性区域中执行, 例如在 `TransactionScope` 活动中。如果在非持久性作用域中使用 `Persist` 活动, 则在运行时 would 引发异常。
- ❑ **TerminateWorkflow 活动**: 终止工作流的执行。

5. 基元

基元活动可用来构建执行常用操作的工作流, 如图 29.13 所示是内置基元活动。



- ❑ Assign 活动：为变量或参数赋值。
- ❑ Delay 活动：可将工作流的执行延迟指定时长。
- ❑ InvokeMethod 活动：调用指定对象或类型的公共方法。
- ❑ Writeline 活动：将文本写入指定的 TextWriter 对象。如果未指定 TextWriter，则 Writeline 会将文本写入控制台中。

6. 事务

使用事务活动，可以将工作流中的某一组活动按原子方式来工作，即这一组活动同时成功或同时失败。这就是事务处理的典型特征。如图 29.14 所示是内置处理事务的活动。



图 29.14 事务

- ❑ CancellationScope 活动：使用 CancellationScope 活动可指定要执行的活动以及该活动的取消逻辑。
- ❑ CompensableActivity 活动：定义可在成功完成之后得到确认或补偿的工作单元。
- ❑ Compensate 活动：为 CompensableActivity 中包含的活动显式调用 CompensationHandler。
- ❑ Confirm 活动：为 CompensableActivity 中包含的活动显式调用 ConfirmationHandler。
- ❑ TransactionScope 活动：在单个事务中执行所包含的活动。在 Body 活动以及该事务中的所有其他参与者成功完成时将提交该事务。

7. 集合

使用集合活动，可以在工作流操作业务数据，并且支持泛型集合。如图 29.15 所示为同内置处理集合的活动。

- ❑ AddToCollection<T>活动：向集合添加一项。
- ❑ ClearCollection<T>活动：清除指定集合中的所有项。
- ❑ ExistsInCollection<T>活动：确定特定集合中是否存在指定项。
- ❑ RemoveFromCollection<T>活动：从特定集合中移除指定项。



图 29.15 集合

8. 错误处理

使用错误处理活动，可在工作流中添加 try/catch/throw 逻辑。如图 29.16 所示是内置的进行错误处理的活动。



图 29.16 错误处理

- ☐ Rethrow 活动：引发先前已引发的异常。此活动只能在 TryCatch 活动的 Catch 处理程序中使用。
- ☐ Throw 活动：引发一个异常。
- ☐ TryCatch 活动：包含由工作流运行时在异常处理块中执行的工作流元素，包含一个 Try 活动、一个 Catch<TException>Try 集合和一个活动。

29.2.3 自定义的代码活动

从前面的介绍可以看出，.NET 4 中内置活动很多，但在实际工作流开发中，仅使用这

些内置活动是完全不够的。这时，就需要使用 WF API 来创建自定义活动。其实现方式比较简单，只需要编写一个继承自 `CodeActivity` 的类即可。如果自定义的活动需要返回值，则创建一个继承自 `CodeActivity<T>` 的类即可（其中 `T` 为返回值类型）。

例如，为前面例子中的工作流再添加功能，如果用户输入的用户名不为 `Test`，则将其输入的用户名和输入的时间保存到一个文本文件中。

显然，使用内置的活动无法完成该功能，这时，可编写一个 `SaveLogActivity` 类来实现保存文件的功能。具体操作步骤如下所述。

（1）在解决方案资源管理器的项目名称上单击鼠标右键，选择“添加”|“新建项”命令打开如图 29.17 所示对话框。



图 29.17 添加新项

（2）在左侧的“已安装的模板”选项中选择 `Visual C#/Workflow`，在右侧的模板中选择“代码活动”，在下方输入新建活动的名称为 `SaveLogActivity.cs`，然后单击“确定”按钮，将创建继承自 `CodeActivity` 类的新类。具体代码如下所示：

```
public sealed class SaveLogActivity : CodeActivity
{
    //定义一个字符串类型的活动输入参数
    public InArgument<string> Text { get; set; }

    //如果活动返回值，则从 CodeActivity<TResult>
    //派生并从 Execute 方法返回该值
    protected override void Execute(CodeActivityContext context)
    {
        //获取 Text 输入参数的运行时值
        string text = context.GetValue(this.Text);
    }
}
```

从自动生成的代码中可看到，用 `InArgument<T>` 类型的属性来表示自定义活动需要输入的内容。`InArgument<T>` 类是 WF API 特有的实体，可以将工作流提供的数据传入自定义活动类的内部。如果需要输入多个参数，可定义多个该类型的属性。

最主要的，在生成的类中需要重写虚方法 `Execute`。当在工作流中执行到自定义活动时，

WF 运行时会调用 Execute 方法。在 Execute 方法中可使用 InArgument<T> 属性，但需使用 CodeActivityContext 的 GetValue 方法来获取参数的值。

(3) 根据以上介绍在自动生成的代码框架中编写代码，具体代码如下：

```
public sealed class SaveLogActivity : CodeActivity
{
    //错误的用户名
    public InArgument<string> UserName { get; set; }

    protected override void Execute(CodeActivityContext context)
    {
        //获取错误用户名
        string userName = context.GetValue(this.UserName);
        System.IO.File.WriteAllText("log.txt",
            userName + ", " + DateTime.Now.ToString());
        Console.WriteLine("用户名错误，已记录到日志文件中！");
    }
}
```

(4) 编写好自定义活动的相关代码后，编译工作程序。然后切换到 workflow 设计器界面，在工具栏中可看到如图 29.18 所示的自定义活动。其中分组名为当前项目名称，而分组中活动的名称就是上面创建的类名 SaveLogActivity。

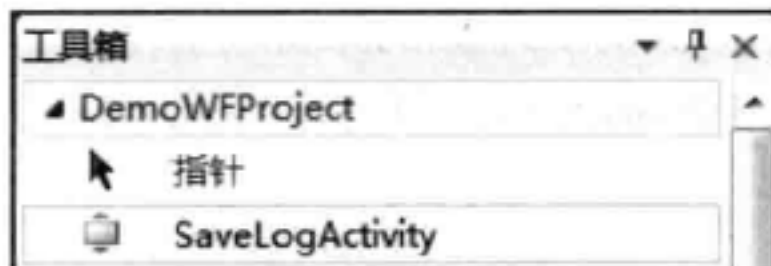


图 29.18 自定义活动

(5) 在工作流设计器中双击打开 If 设计界面，将 Else 中原来的 WriteLine 删除，重新从工具箱中拖入自定义的 SaveLogActivity，并在“属性”窗口设置 UserName 为工作流中定义的变量 UserName 的值，如图 29.19 所示。

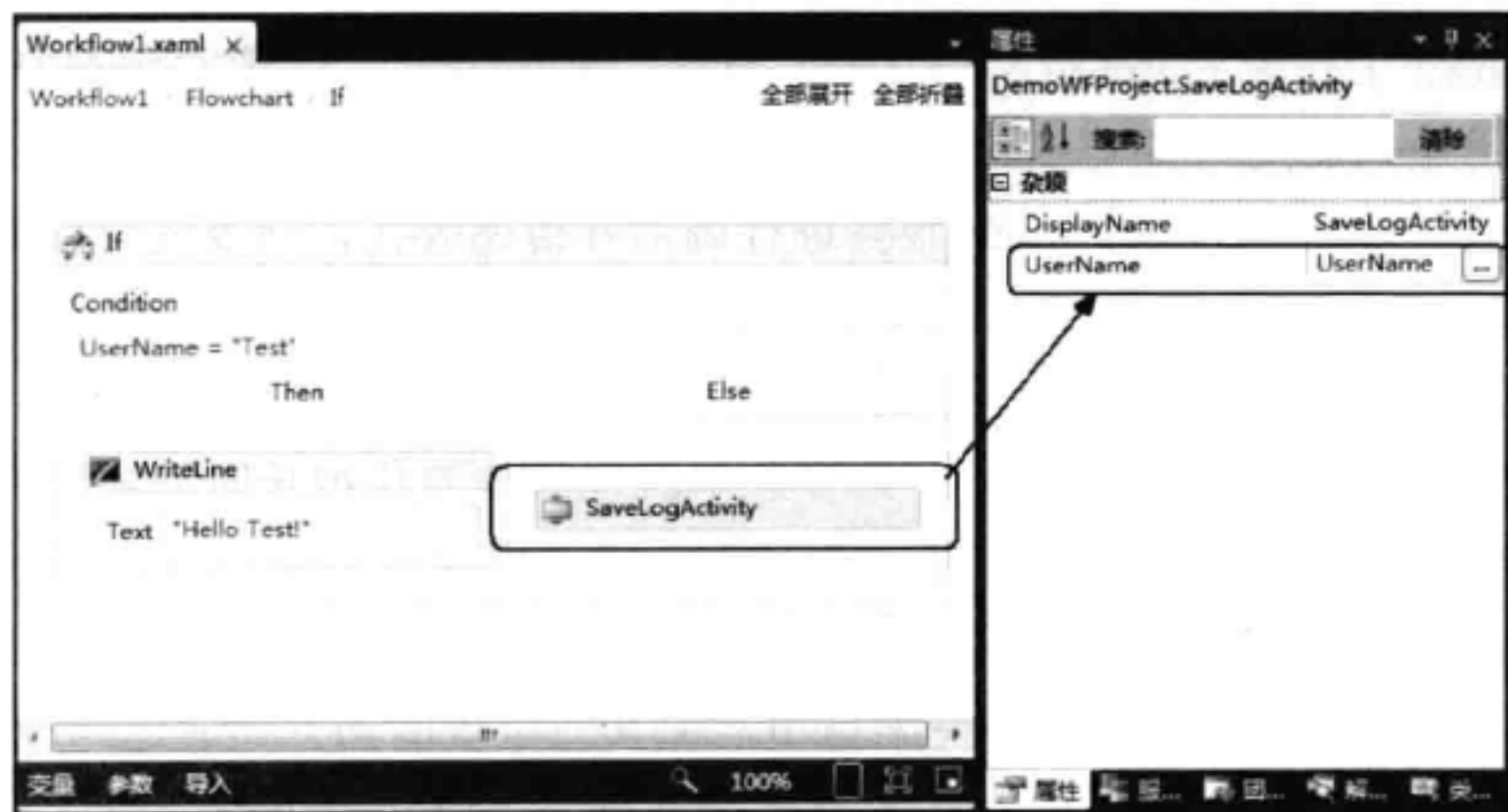


图 29.19 使用自定义活动

(6) 启动 workflow 应用程序，输入 abc（不为 Test 即可），可看到屏幕上的提示，如图 29.20 中的左图所示，此时在当前程序文件夹中（bin\Debug）可看到有一个名为 log.txt 的

文件，打开该文件可看到如图 29.20 中的右图所示内容。




图 29.20 执行结果

从以上修改工作流的过程还可看出，虽然对工作流中的活动进行了调整，但并不需要程序员修改原有程序代码中的内容。也就是说，当将应用程序开发完成后，如果要调整工作流，只需要在工作流设计界面进行调整即可，而其调用程序是不需要进行改变的。这也是在应用程序中使用工作流的一个优势。

29.2.4 WF 提供的服务

WF 提供了多项服务，应用程序可以使用这些服务来执行事务处理工作、流程的管理工作、工作流实例的数据保存，以及工作流的跟踪等操作。WF 包含 Windows 工作流计划服务、Windows 工作流 CommitWorkBatch 服务、Windows 工作流持久性服务和 Windows 工作流跟踪服务等几种服务。

 **说明：**WF 服务提供了对工作流运行过程中的辅助功能，可以使用相关的服务，实现对工作流扩展功能的开发。

- ❑ **Windows 工作流计划服务：**Windows 工作流计划服务管理工作流运行时引擎计划工作流实例的方式。无论这些服务是通过异步方式处理，还是通过以手动、同步的方式处理，它们都是工作流解决方案的重要部分。
- ❑ **Windows 工作流 CommitWorkBatch 服务：**Windows 工作流 CommitWorkBatch 服务的用途是启用与提交工作批次相关的自定义逻辑。当提交工作批次时，运行时将对当前 CommitWorkBatch 服务进行调用并传递委托，该委托可以对工作批次执行实际提交。运行时仍必须执行提交，但是它允许服务将自身插入到进程中，从而支持针对提交进程进行一些自定义。
- ❑ **Windows 工作流持久性服务：**许多业务流程都需要花很长时间才能完成，将工作流保存在内存中不仅不切实际，而且因为必须在单一服务器上处理实例，所以还会妨碍缩放。许多这些长期运行的工作流都是执行不活跃的流程或过程逻辑，并且实际上处于空闲状态，等待来自用户或其他系统的输入。通过卸载空闲的实例，主机应用程序将能节省内存，并且能跨处理服务器进行缩放。当工作流运行的时候发生特定情况时，工作流运行时引擎就会使用持久性服务保留有关工作流实例的状态信息。
- ❑ **Windows 工作流跟踪服务：**使用 WF 可以以一致、可靠而灵活的方式跟踪与工作

流相关的信息。WF 跟踪框架旨在使宿主通过捕获 workflow 执行期间引发的事件，而在执行期间可以观察到 workflow 实例。

29.3 WF 创建工作流实例

WF 可以创建两种基本类型的工作流，一个是顺序工作流，另外一种状态机工作流。本节通过几个完整的应用实例，介绍如何使用 Visual Studio 2010 创建工作流的应用。

29.3.1 在工作流中使用集合

在 29.2.2 节中介绍了系统内置的 WF 活动，其中包含一个名为“集合”的组，其中提供的活动可用来对工作流中的集合进行处理。下面以一个例子来演示在工作流中使用集合的方法。

这里假设用一个工作流处理人员名单，为简化例子，在工作流中将人员名单添加到集合中，再输出集合中人员的信息，接着清空集合，然后重新添加人员名单到集合，再输出集合中的信息。

完成以上工作流设计的操作步骤如下所述。

(1) 在 Visual Studio 2010 中新建一个控制台工作流应用程序 WFCollection。

(2) 创建一个保存人员信息的类。在解决方案资源管理器中右击项目名称，从弹出的菜单中选择“添加”|“新建项”命令，向项目添加一个名为的 Person 类。具体代码如下：

```
class Person
{
    private string Name { set; get; }
    private string Age { set; get; }
    private string Sex { set; get; }

    public Person(string sName, string sSex, int iAge)
    {
        this.Name = sName;
        this.Sex = sSex;
        this.Age = iAge;
    }

    public override string ToString()
    {
        return Name + " , " + Sex + " , " + Age.ToString();
    }
}
```

(3) 创建一个输出人员信息的活动。在解决方案资源管理器中右击项目名称，从弹出的菜单中选择“添加”|“新建项”命令，打开“添加新项”对话框，在“已安装的模板”选项中选择 Visual C# / Workflow，在右侧模板中选择“代码活动”，输入名称 PrintPerson，如图 29.21 所示。



图 29.21 添加新项

(4) 在新添加的自定义活动 `PrintPerson` 中编写以下代码：

```
public sealed class PrintPerson<Person> : CodeActivity
{
    public InArgument<ICollection<Person>> Collection { get; set; }

    protected override void Execute(CodeActivityContext context)
    {
        ICollection<Person> persons =
            this.Collection.Get<ICollection<Person>>(context);
        if (persons.Count == 0)
        {
            Console.WriteLine("集合为空");
        }
        else
        {
            foreach (Person p in persons)
            {
                Console.WriteLine(p.ToString());
            }
        }
    }
}
```

(5) 编译项目，然后切换到工作流设计界面，在“工具箱”中可看到新建的活动 `PrintPerson<T>`。

(6) 将以上准备工作做完之后，就可以设计工作流了。在“工具箱”的“控制流”组中将 `Sequence` 活动拖到设计器中并使其处于选中状态，然后单击设计器下方的“变量”打开变量设置面板，在“名称”中输入 `Persons`，在“变量类型”中单击下拉列表框选择“浏览类型”，打开“浏览并选择 .NET 类型”对话框，在上方输入查找的类型 `System.Collections.ObjectModel.Collection`，这时下方列表框中只显示这一个类型。由于选择的是一个泛型类型，还需要继续操作，单击输入框下方的下拉列表框，从中选择 `WFCollection.Person`，如图 29.22 所示。

(7) 通过以上步骤即可创建一个名为 `Persons` 的变量，用该变量来保存集合中的数据，接着在“默认值”中输入以下代码：


```
New System.Collections.ObjectModel.Collection(Of WFCollection.Person)
From {}
```



图 29.22 添加新项

注意，这里是用 VB.NET 的语法表示默认值，以上语句设置默认值为空。设置好的“变量”面板如图 29.23 所示。

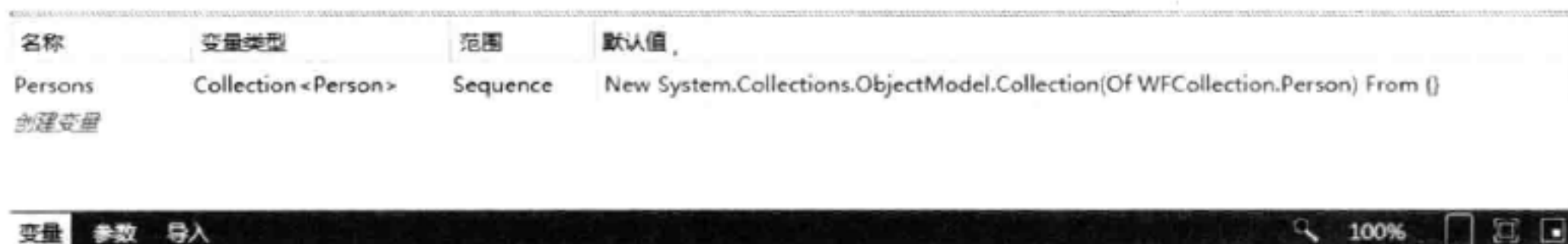


图 29.23 添加变量

(8) 从“工具箱”的“基元”组中拖动 WriteLine 活动到设计器中的 Sequence 活动中，然后在“属性”窗口中设置其 DisplayName 属性为“显示提示信息”（这里显示在设计器中当前活动标题中的内容），Text 属性为“首先添加 3 个人员信息到集合中。”（这是输出的内容），如图 29.24 所示。



图 29.24 添加 WriteLine 活动

(9) 从“工具箱”的“集合”组中拖动一个 `AddToCollection<T>` 活动到 `WriteLine` 活动下方,在“属性”窗口中设置其 `DisplayName` 属性为“添加一人的信息”,设置 `TypeArgument` 属性为 `WFCollection.Person` (可单击下拉箭头进行选择),设置 `Collection` 属性为 `Persons` (图 29.23 中定义的变量),接着单击 `Item` 右侧的省略号按钮打开如图 29.25 所示的“表达式编辑器”对话框,输入一个人员的信息。



图 29.25 表达式编辑器

(10) 对 `AddToCollection<T>` 活动设置完成后,可看到如图 29.26 所示的效果。



图 29.26 添加 `AddToCollection<T>` 活动

(11) 重复前面的步骤,再向工作流中添加两个 `AddToCollection<T>` 活动,当然在图 29.25 所示的对话框中要输入不同的值,以表示不同的人员信息。

(12) 从“工具箱”的“`WFCollection`”组(自定义活动所在组)中将自定义的名为 `Printperson<T>` 的活动拖到工作流中,用来打印输出人员信息。这时,将弹出如图 29.27 所示的“选择类型”对话框,从下拉列表框中选择“`WFCollection.Person`”。接着在“属性”窗口中设置 `Collection` 为 `Persons`,即可将变量中的值传递到自定义活动。

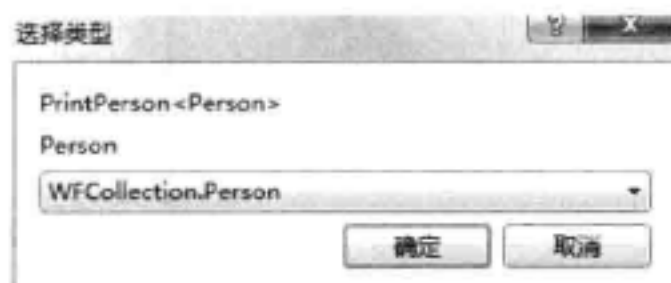


图 29.27 选择类型

(13) 从“工具箱”的“集合”组中拖动一个 `ClearCollection<T>` 活动到工作流的下方,在“属性”窗口中设置其 `DisplayName` 属性为“清除集合中的内容”,`Collection` 属性为

Persons, TypeArgument 属性为 WFCollection.Person。

(14) 再将 Printperson<T>活动拖到 workflow 最下方, 进行类似设置。至此, 完成了 workflow 的设计, 得到如图 29.28 所示的 workflow 设计图。

(15) 由于在 Program 类中已定义了调用 workflow 的代码, 因此可以直接运行项目得到 workflow 执行结果, 如图 29.29 所示。

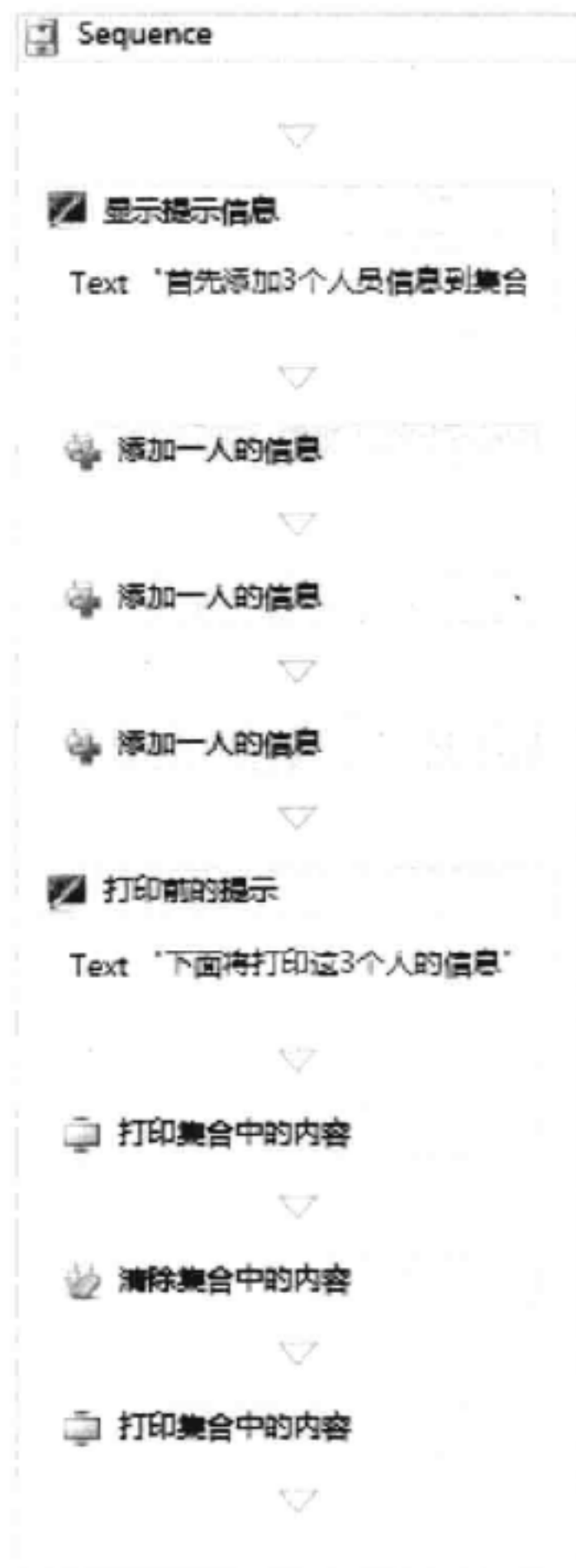


图 29.28 workflow 设计图

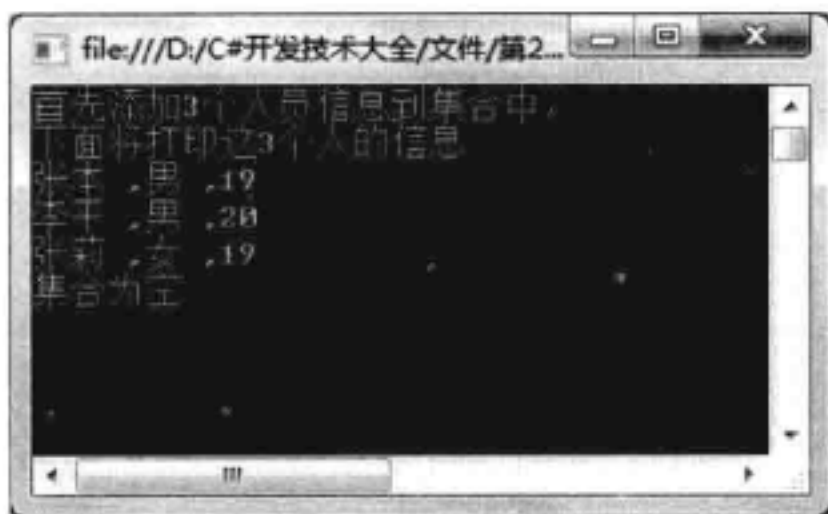


图 29.29 workflow 执行结果

29.3.2 猜价格游戏

上一个例子比较简单, 整个 workflow 按照顺序方式执行。在实际应用中, 很多 workflow 都有分支、循环等流程。下面以一个简单的猜价格游戏来演示这些流程的用法。

猜价格游戏的过程: 假设有一件商品, 其价格在 100~1000 元之间, 参与者首先猜一个价格, 比较其与商品实际价格给出太高、太低或猜中的提示, 如果没有猜中, 则参与者可重复给出价格, 直到猜中为止, 最后看谁猜中时重复的次数少。

分析以上游戏, 可看出既有三重分支, 又有循环。用 workflow 创建该游戏的过程如下

所述。

- (1) 在 Visual Studio 2010 中新建一个名为 WFLuck 的控制台 workflow 应用程序。
- (2) 向 workflow 设计视图中拖入一个 Sequence 活动，修改 DisplayName 属性为“幸运猜价格”。
- (3) 单击选中 Sequence 活动，再单击下方的“变量”标签定义如图 29.30 所示的 4 个变量。其中 Price 为商品的单价，input 为参与者输入的单价（字符串类型），finished 若为 true 表示已猜中价格，attempts 为重复试猜的次数。

名称	变量类型	范围	默认值
Price	Int32	幸运猜价格	输入 VB 表达式
input	String	幸运猜价格	输入 VB 表达式
finished	Boolean	幸运猜价格	false
attempts	Int32	幸运猜价格	0
创建变量			
变量 参数 导入			

图 29.30 定义变量

- (4) 向 Sequence 活动中拖入一个 Assign 活动，用来随机设置商品的价格，设置其属性：DisplayName 为“商品价格”，To 为 Price，Value 为 New System.Random().Next(100, 1000)。
- (5) 向 workflow 下方拖入一个 While 活动，用来循环猜价，设置其属性：DisplayName 为“循环猜价”，Condition 为 Not finished（即 finished 变量为 false 则进行循环）。得到的 workflow 设计图如图 29.31 所示。



图 29.31 添加 While 活动

- (6) 向 While 活动的 Body 部分拖入一个 Sequence 活动。
- (7) 向内层 Sequence 活动中拖入一个 Assign 活动，用来增加试猜的次数，如图 29.32 所示。

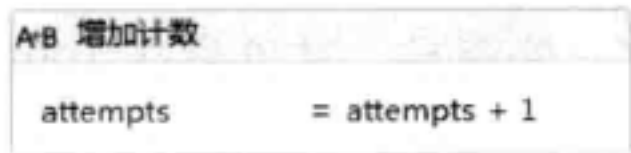


图 29.32 添加 Assign 活动

(8) 在 Assign 下方拖入一个 TryCatch 活动, 如图 29.33 所示。

(9) 向 TryCatch 活动的 Try 部分拖入一个 Sequence 活动。

(10) 向最内层 Sequence 活动中拖入一个 WriteLine 活动, 用来向控制台输入一个提示信息“请输入您猜的价格:”。

(11) 在 WriteLine 下方拖入一个 InvokeMethod 活动, 用来调用 System.Console.ReadLine 方法, 设置其 Result 属性为 input, 即将输入的内容保存到 input 变量中, 如图 29.34 所示。



图 29.33 添加 TryCatch 活动



图 29.34 添加 InvokeMethod 活动

(12) 向 InvokeMethod 活动下方拖入一个 If 活动, 用来判断参与者输入的价格是否在规定的区间, 设置其条件如图 29.35 所示。



图 29.35 添加 If 活动

(13) 向 If 活动的 Then 部分拖入一个 Throw 活动, 用来抛出一个异常供 TryCatch 捕获。设置 Throw 活动的 Exception 属性为 New OverflowException(), 即抛出一个溢出错误。

(14) 向 If 活动下方拖入一个 Switch<T>活动, 用来比较参与者输入的价格与商品的实际价格。设置其 Expression 为“Int32.Parse(input).CompareTo(Price)”, 输入的值保存在 input 变量中, 将其转换为 Int32 类型再与商品价格 Price 进行比较。比较结果有 3 种情况, 在 Switch<T>中单击“添加新事例”文字添加 3 种情况, 分别设置 Case 为 1、-1、0。在 Case 为 1 的事例中添加一个 WriteLine 活动, 向控制台输出“太高...”, 如图 29.36 中的左图所示。

(15) 类似的, 向 Case 为 -1 的事例中添加一个 WriteLine 活动, 输出“太低...”。

(16) 类似的, 向 Case 为 0 的事例中添加一个 Assign 活动, 将 finished 变量赋值为 true, 如图 29.36 中的右图所示。



图 29.36 设置 Switch<T>活动的 Case 事例

(17) 返回到整个工作流设计界面, 在 TryCatch 活动的 Catches 部分单击“添加新捕获”文字, 将出现一个选择 Exception 的下拉列表框, 从中找到 OverflowException 异常, 设置好之后单击鼠标。然后向 OverflowException 中拖入一个 WriteLine 活动, 用来向控制台输出一个错误信息, 如图 29.37 所示。

(18) 最后, 在最外层 Sequence 中的最下方拖入一个 WriteLine 活动, 设置其 Text 属性如图 29.38 所示, 即向控制台显示共重试了多少次才猜中价格。



图 29.37 捕获异常

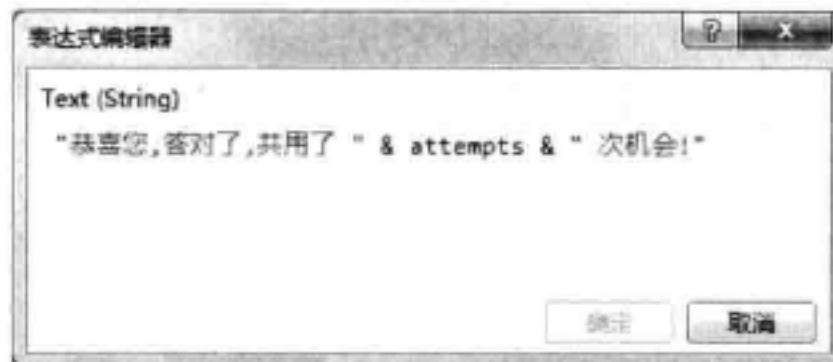


图 29.38 表达式编辑器

通过以上步骤, 就将整个猜价格游戏的流程设计完成了。由于整个流程图比较大, 一次显示不完, 为了让读者对流程有个整体印象, 图 29.39 列出了外层流程。



图 29.39 外层流程

从图中可看到整个流程有 3 大部分, 第一是设置商品价格, 第二是猜价, 第三是输出。

而最重要的就是循环猜价部分,这部分根据变量 `finished` 的值判断是否循环,若其值为 `false`,则循环猜价。循环猜价部分的流程如图 29.40 所示,这部分首先增加试猜次数,然后让参与者输入价格,并在流程中进行比较,以判断是否猜中。



图 29.40 循环猜价的流程

对工作流的执行流程了解之后,按 F5 键执行该流程,可得到如图 29.41 所示的猜价效果。其中有一次特别输入 0,可看到提示“只能输入 100~1000 之间的整数!”。



图 29.41 流程执行效果

29.4 本章总结

工作流是以流程的方式进行业务处理的一种通用的应用程序。工作流的内容很广泛，需求也复杂多样，基于 WF 工作流框架可以实现灵活的流程设计需求，原因就是 WF 集成在 .NET 框架之中，使得其开发部分能足够灵活，以满足复杂的流程需求。通过本章的学习，读者可以掌握基本的 WF 框架概念、框架元素，以及在 Visual Studio 2010 中创建简单的工作流应用。

29.5 实战练习

1. 在 Visual Studio 2010 中新建一个控制台工作流应用程序，创建工作流完成以下任务：由用户在控制台输入 100 分制的成绩，根据这些成绩分别输出 A、B、C、D、E 5 个等级（60 分以下为 E，60~70 分为 D，70~80 分为 C，80~90 分为 B，90 分以上为 A）。
2. 在 Visual Studio 2010 中新建一个控制台工作流应用程序，创建工作流完成以下任务：向一个集合中添加 5 个整数，并将这 5 个整数输出到控制台。
3. 在 Visual Studio 2010 中新建一个控制台工作流应用程序，创建工作流完成以下任务：用户在控制台输入需要生成随机数的数量，然后通过 While 活动生成指定数量的随机数，并将这些随机数保存到集合中，最后输出生成的所有随机数。

第 30 章 语言集成查询 LINQ


Language Integrated Query 简称 LINQ，是微软在新一代的框架体系中提供的一种语言集成查询框架。通过强大的 LINQ 查询功能，开发人员可以在对象领域和数据领域建立起一座桥梁。通过强大的查询语言，完成对数据库、数据集、XML，以及自定义对象的数据查询。本章将从 LINQ 的概述、语法、应用，以及最终的应用场景来介绍 LINQ 技术。

30.1 LINQ 概述

查询是通过查询表达式完成对数据源的数据检索功能，在 .NET 4 框架中，使用了 LINQ 作为其查询语言，可以实现对数据源的查询功能。

30.1.1 了解 LINQ 查询

LINQ 通过提供一种跨各种数据源和数据格式使用数据的一致模型，使得在框架中进行查询实现起来更方便。在 LINQ 查询中，始终会用到对象。可以使用相同的基本编码模式来查询和转换 XML 文档、SQL 数据库、ADO.NET 数据集、.NET 集合中的数据，以及对其有 LINQ 提供程序可用的任何其他格式的数据。

 **技巧：**在 LINQ 查询框架中，对象的查询创建基本需要 3 个部分：获取数据源、创建查询语句，以及执行查询。

获取数据源是指首先定义查询目标的数据源类型，数据源可以是数据库、数据集、XML 文件或者自定义的对象等。无论是何种数据源，首先需要获取该数据源，便于接下来完成对该数据源的查询定义。

创建查询语句是指通过通用的 LINQ 查询语法，定义本次查询的内容。除了查询数据之外，查询语句还可以指定在返回这些信息之前如何对其进行排序、分组和结构化。查询存储在查询变量中，并用查询表达式进行初始化。为使编写查询的工作变得更加容易，C# 引入了新的查询语法。

执行查询是指最终的查询语句的运行。查询变量本身只是存储查询命令，实际的查询执行会延迟在查询执行语句中循环访问查询变量时发生。由于查询变量本身从不保存查询结果，因此可以根据需要随意执行查询。例如，可以通过一个单独的应用程序持续更新数据库。在应用程序中，可以创建一个检索最新数据的查询，并可以按某一时间间隔反复执行该查询，以便每次检索不同的结果。

30.1.2 简单 LINQ 查询实例

LINQ 查询是一种通用的查询语法，它可以在程序的任意逻辑中使用。本节将通过一个控制台应用程序来实现 LINQ 查询的应用介绍。

(1) 创建一个控制台应用程序，命名为 LINQAppDemo。

(2) 在控制台的主控制类 Program.cs 中的 Main()函数中，添加如下代码：

```
static void Main(string[] args)
{
    //定义数据源
    int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

    //创建查询语句
    var numQuery =
        from num in numbers //指定查询对象
        where (num % 2) == 0 //指定查询条件
        select num;         //选择查询结果

    //执行查询
    foreach (int num in numQuery)
    {
        //显示查询结果
        Console.WriteLine("{0,1} ", num);
    }
    Console.ReadLine();
}
```

在上面的代码中，首先定义了一个整型的数组对象 `numbers`，以此作为数据源进行后续的查询操作。然后创建查询语句，通过 `var` 定义了一个查询语句变量 `numQuery`。`from` 语句后续定义了一个查询临时变量，用于定义数据源中的查询临时数据。`in` 语句指定了查询的数据源对象为 `numbers`。`where` 语句后续定义了查询的条件，表示该查询结果必须满足能够被 2 整除。最后通过 `select` 语句将符合该条件的数据全部选择出来，再通过 `foreach` 语句变量查询变量对象 `numQuery`，执行查询，将符合条件的查询结果显示到控制台中。

(3) 运行该实例，最终的查询结果如图 30.1 所示。

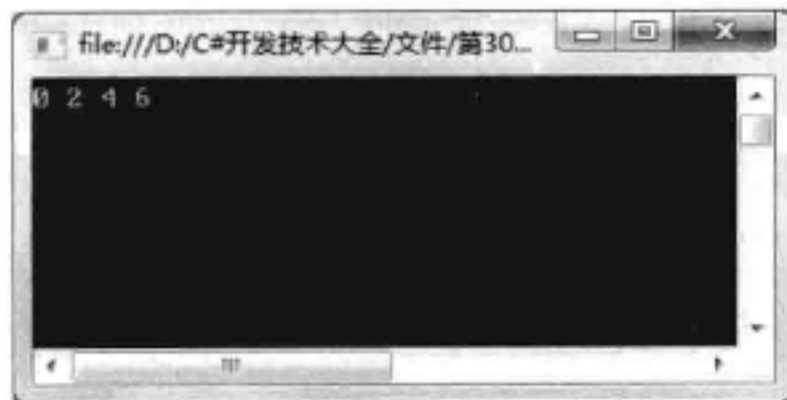


图 30.1 简单 LINQ 查询结果

30.2 LINQ 语言基础


语言集成查询是一组技术的名称，这些技术建立在将查询功能直接集成到 C#语言的基础

础上。借助于 LINQ，查询已是高级语言构造，就如同类、方法、事件等一样。

30.2.1 最重要的 LINQ 查询表达式

对于编写查询的开发人员来说，LINQ 最重要的部分是查询表达式。查询表达式是使用 C# 中引入的声明性查询语法编写的。通过使用查询语法，可以使用最少的代码对数据源执行复杂的筛选、排序和分组操作。可以使用相同的基本查询表达式模式来查询和转换 SQL 数据库、ADO.NET 数据集、XML 文档和流，以及 .NET 集合中的数据。

查询表达式是用查询语法表示的查询，是一种方便的语法。它就像其他表达式一样，并且可以用在 C# 表达式有效的任何代码中。查询表达式由一组用类似于 SQL 或 XQuery 的声明性语法编写的子句组成。每个子句又包含一个或多个 C# 表达式，而这些表达式本身又可能是查询表达式或包含查询表达式。

 **说明：**查询表达式必须以 from 子句开头，并且必须以 select 或 group 子句结尾。在第一个 from 子句和最后一个 select 或 group 子句之间，查询表达式可以包含一个或多个可选子句，如 where、orderby、join、let，甚至附加的 from 子句。还可以使用 into 关键字使 join 或 group 子句的结果能够充当同一查询表达式中附加查询子句的源。

查询表达式有以下几个方面的优点：

- ❑ 查询表达式可用于查询和转换来自任意支持 LINQ 的数据源中的数据。
- ❑ 查询表达式容易掌握，因为它们使用许多常见的 C# 语法。
- ❑ 查询表达式中的变量都是强类型的，但许多情况下不需要显式提供类型，因为编译器可以推断类型。
- ❑ 任何可以使用查询语法表示的查询也可以使用方法语法表示。但是，在大多数情况下，查询语法更简洁易读。
- ❑ 作为编写 LINQ 查询的一项规则，建议尽量使用查询语法，只在必需的情况下才使用方法语法。

下面的代码示例演示了一个简单的查询表达式，它含有一个数据源、一个筛选子句和一个排序子句，但不对源元素进行转换，select 子句结束了该查询。

```
//数据源
int[] scores = { 90, 71, 82, 93, 75, 82 };

//查询表达式
IEnumerable<int> scoreQuery = //查询变量
    from score in scores      //必需的
    where score > 60          //可选的
    orderby score descending  //可选的
    select score;             //必须以 select 语句或者 group 语句结尾

//执行查询，产生查询结果
foreach (int test in scoreQuery)
{
    Console.WriteLine(test); //显示查询结果
}
```

```
}
```

查询结果的产生可以通过 `foreach` 语句进行遍历,也可以通过 `ToList()`方法进行转换,形成新的集合类型之后再进行处理,如下代码所示。

```
IEnumerable<City> largeCityList =
    (from country in countries           //指定查询对象
     from city in country.Cities        //指定查询对象
     where city.Population > 10000      //指定查询条件
     select city)                       //选择查询结果
    .ToList();                          //执行转换
```

`largeCityList` 对象就是通过查询执行结果之后的 `ToList()`方法,形成了一个 `City` 对象的集合。

除了可以返回所有的查询结果列表之外,还可以实现对结果的一个统计功能,比如查询前面的所有分数中的最高得分,可以通过 `Max` 方法来获取到,代码如下:

```
int highestScore =
    (from score in scores               //指定查询对象
     select score)                     //选择查询结果
    .Max();                            //求最大值
```


`Select` 语句是用来将结果选择出来,还可以通过 `group by` 语句实现对结果进行分组。代码如下:

```
var queryCountryGroups =
    from country in countries           //指定查询对象
    group country by country.Name[0];  //分组查询结果
```

如果想实现对查询结果的排序处理,就需要使用 `order by` 语句,并且结合正序或者倒序参数来实现,代码如下:

```
IEnumerable<Country> querySortedCountries =
    from country in countries           //指定查询对象
    orderby country.Area > 500000,      //指定查询条件
    country.Population descending       //查询结果排序
    select country;                     //选择查询结果
```

其中的 `descending` 参数表示为倒序排列结果。

 **注意:** 如果想获得正序的结果,可以使用参数 `ascending`,或者默认不添加排序参数,因为排序默认就采用正序的排序结果。

使用 `join` 子句可以根据每个元素中指定键之间的相等比较,对一个数据源中的元素与另外一个数据源中的元素进行关联和/或组合,代码如下:

```
var categoryQuery =
    from cat in categories               //指定查询对象
    join prod in products on cat equals prod.Category //联合查询
    select new { Category = cat, Name = prod.Name }; //选择查询结果
```

使用 `let` 子句可以将表达式的结果存储到新的范围变量中,代码如下:

```
string[] names = { "firstName lastName" }; //定义查询数据源
```



```

IEnumerable<string> queryFirstNames =
    from name in names                //指定查询对象
    let firstName = name.Split(new char[] { ' ' })[0] //指定查询条件
    select firstName;                 //选择查询结果

```

30.2.2 LINQ 查询语法和方法语法实例

LINQ 的查询包括两种基本的语法，即查询语法和方法语法。查询语法是指在查询中使用 `select` 语句直接返回查询列表，而方法语法是对于查询结果的一些统计功能，如求和 `Sum`、求最大值 `Max`、求最小值 `Min`、求平均值 `Average` 等。在查询中，二者可以独立使用，也可以结合起来一起使用。

在 30.2.1 节中介绍了查询语法和方法语法的一些使用方式。本节将通过一个完整的实例介绍如何使用查询语法和方法语法进行 LINQ 的查询。

(1) 创建一个控制台项目，命名为 `QueryMethodDemo`。

(2) 添加查询语法代码。在 `Program.cs` 添加如下代码：

```

//查询 1
//定义数据源
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
//定义查询变量
IEnumerable<int> filteringQuery =
    from num in numbers                //指定查询对象
    where num < 3 || num > 7          //指定查询条件
    select num;                       //选择查询结果

Console.WriteLine("查询 1 结果: ");
foreach (int intTemp in filteringQuery)
{
    Console.WriteLine(intTemp); //显示查询结果
}

//查询 2, 定义查询条件
IEnumerable<int> orderingQuery =
    from num in numbers                //指定查询对象
    where num < 3 || num > 7          //指定查询条件
    orderby num ascending              //排序
    select num;                       //选择查询结果

Console.WriteLine("查询 2 结果: ");
//变量查询结果
foreach (int intTemp in orderingQuery)
{
    Console.WriteLine(intTemp); //显示查询结果
}

//查询 3
//定义数据源
string[] groupingQuery = { "apple", "banana", "orange", "pear",
    "strawberry" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery        //指定查询对象

```



```

        group item by item[0];           //分组

        Console.WriteLine("查询 3 结果: "); //显示查询结果
        //遍历查询结果
        foreach (IGrouping<char, string> mi in queryFoodGroups)
            Console.WriteLine(mi.Key);

```

该部分 3 个查询全都是通过查询语法创建的查询表达式。第 1 个查询表达式演示如何通过用 **where** 子句应用条件来筛选或限制结果，它返回源序列中值大于 7 或小于 3 的所有元素。第 2 个表达式演示如何对返回的结果进行排序。第 3 个表达式演示如何按照键对结果进行分组，此查询可根据单词的第一个字母返回两个组。

(3) 添加方法语法代码。接着再添加以下代码：

```

        //定义数据源
        List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
        List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };

        //查询 4，求数据源的平均值
        double average = numbers1.Average();           //求平均值
        Console.WriteLine("查询 4 结果: ");             //显示查询结果
        Console.WriteLine("average: " + average.ToString()); //显示查询结果

        //查询 5
        //连接数据源数据
        IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
        Console.WriteLine("查询 5 结果: ");
        foreach (int intTemp in concatenationQuery)
        {
            Console.WriteLine(intTemp);                 //显示查询结果
        }

        //查询 6
        //定义查询变量
        IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
        Console.WriteLine("查询 6 结果: ");
        foreach (int intTemp in largeNumbersQuery)
        {
            Console.WriteLine(intTemp);                 //显示查询结果
        }

```

该部分的 3 个查询全部基于方法语法进行查询。查询 4 使用 **Average()** 方法获取数据源的平均值。查询 5 通过 **Concat()** 方法将两个数据源进行连接查询。查询 6 通过 **Where()** 方法的 **lambda** 表达式进行查询。

(4) 添加方法语法和查询语法混合代码，接下来添加以下代码：

```

//查询 7
//使用方法语法查询结果的数量
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();
//使用查询语法查询结果的数量
IEnumerable<int> numbersQuery =
    from num in numbers1

```



```

        where num < 3 || num > 7
        select num;

        int numCount2 = numbersQuery.Count();

        Console.ReadLine();

```

可以通过代码看出，查询 7 属于查询语法和方法语法的混合查询。Count 使用的是方法语法，获取集合中的数量，而通过 select 语句的查询语法获取小于 3 或者大于 7 的数据集合。

(5) 运行该项目，运行结果如图 30.2 所示。

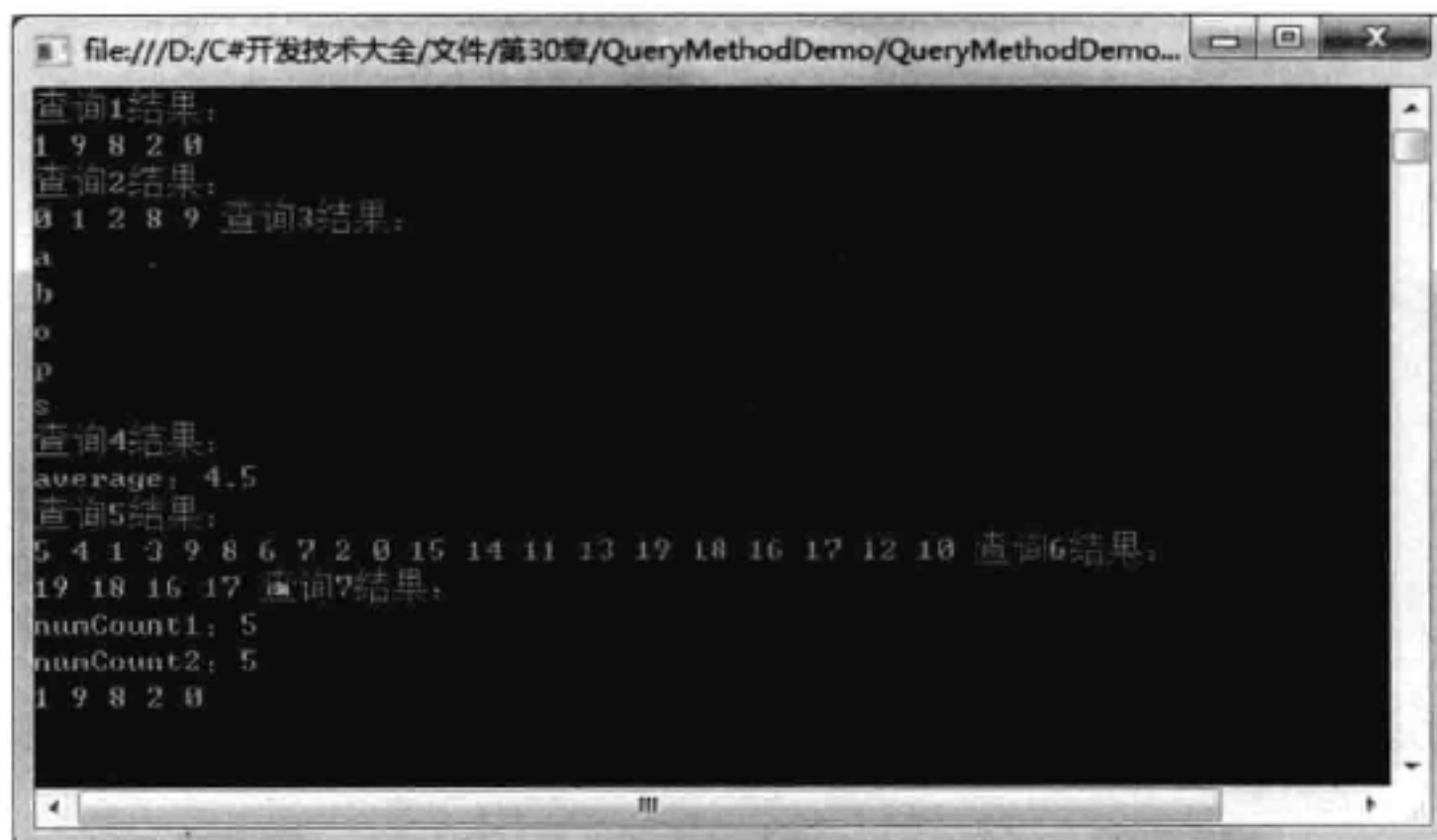


图 30.2 查询语法和方法语法实例运行结果

30.2.3 用 LINQ 合并数据

语言集成查询不仅可用于检索数据，而且还是一个功能强大的数据转换工具。本节将通过实例介绍如何通过 LINQ 实现数据的合并。

- (1) 创建一个控制台应用程序，命名为 MergeDataDmeo。
- (2) 在 Program.cs 文件中，添加两个类的定义代码如下：

```

class A
{
    //定义 Name 属性
    public string Name { get; set; }
    //定义 ID 属性
    public int ID { get; set; }
}

class B
{
    ///定义 Name 属性
    public string Name { get; set; }
    //定义 ID 属性
    public int ID { get; set; }
}

```

这两个类定义的属性完全一样。

(3) 在 Main()方法中, 首先定义第一个容器类的实例, 作为第一个数据源, 代码如下:

```
//创建第一个数据源
List<A> a = new List<A>()
{
    //添加列表数据
    new A {ID=1,
        Name="Tom"},
    new A {ID=2,
        Name="Mike"},
    new A {ID=3,
        Name="John"},
    new A {ID=4,
        Name="Tony"}
};
```

接着创建第二个容器类的实例, 作为第二个数据源, 代码如下:

```
//创建第二个数据源
List<B> b = new List<B>()
{
    new B {ID=1, Name = "Mary"},
    new B {ID=2, Name = "Jane"},
    new B {ID=3, Name = "Kate"}
};
```

最后添加查询代码, 然后通过 Concat 方法将两个数据源满足条件的数据全部查询出来, 代码如下:

```
//创建查询
var personSameID = (from personA in a //指定查询对象
                    where personA.ID == 1 //指定查询条件
                    select personA.Name) //选择查询结果
//连接查询结果
.Concat(from personB in b
        where personB.ID==1
        select personB.Name);

Console.WriteLine("相同 ID 的人:");
//遍历查询结果
foreach (var person in personSameID)
{
    Console.WriteLine(person); //选择查询结果
}

Console.ReadKey();
```

(4) 运行项目, 结果如图 30.3 所示。

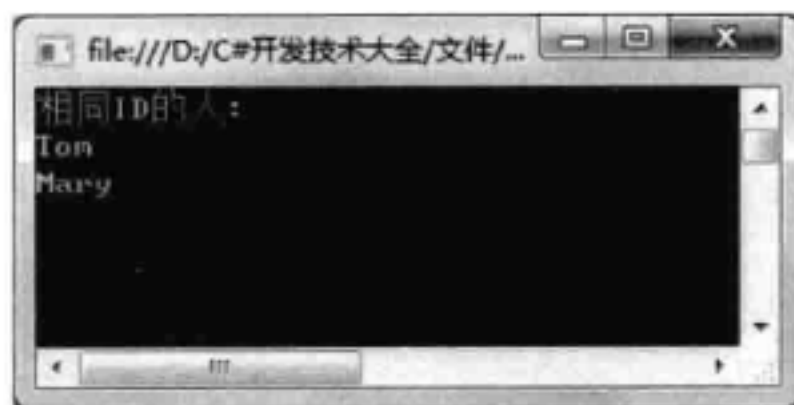


图 30.3 使用 LINQ 进行数据合并实例运行结果

30.2.4 用 LINQ 转换数据

LINQ 不仅可以查询数据、修改查询结果，而且可以用来完成查询之后的数据转换功能，例如它可以将数据源的数据，经过一定的处理，转换成 XML 格式的数据。本节将通过一个实例来介绍如何实现 LINQ 对数据的转换。

(1) 创建一个控制台项目，命名为 ConvertDataDemo。

(2) 在 Program.cs 文件中，添加类的定义。

```
class A
{
    //定义 Name 属性
    public string Name { get; set; }
    //定义 ID 属性
    public int ID { get; set; }
    //定义 Phones 属性
    public List<int> Phones;
}
```

该对象类具有 3 个公有属性，同时第 3 个属性 Phones 还是一个列表整型。

(3) 添加数据源定义，代码如下：

```
List<A> a = new List<A>()
{
    //添加列表数据
    new A {ID=1, Name = "Mary", Phones = new List<int>{1111, 2222, 3333}},
    new A {ID=2, Name = "Jane", Phones = new List<int>{4444, 5555, 6666}},
    new A {ID=3, Name = "Kate", Phones = new List<int>{7777, 8888, 9999}}
};
```

定义了一个 a 集合，集合中添加了 3 个 A 的实例。

添加如下用于 LINQ 转换的代码：

```
//定义查询变量，生成 XML 对象
var AsToXML = new XElement("Root",           //定义根元素名称
    from personA in a                         //指定查询对象
    //格式化查询结果
    let x = String.Format("{0},{1},{2}", personA.Phones[0],
        personA.Phones[1], personA.Phones[2])
    select new XElement("A",                  //选择根元素
        new XElement("ID", personA.ID),      //创建 ID 元素
        new XElement("Name", personA.Name),  //创建 Name 元素
        new XElement("Phones", x)            //创建 Phones 元素
    ) // a 结束"
); // "Root"结束

Console.WriteLine(AsToXML); //显示查询结果

Console.ReadKey();
```

(4) 运行该项目，结果如图 30.4 所示。



图 30.4 使用 LINQ 进行数据转换实例运行结果

30.3 LINQ 查询数据源

LINQ 定义了查询语言，它可以通过对多种数据源对象进行查询，从而实现数据的高级检索功能。这些数据源包括数据库、数据集、XML 对象，以及常用的对象等。本节将通过以上 4 种数据源查询的介绍，使读者可以更深入的了解 LINQ 在简单对象模型中查询的应用。

30.3.1 用 LINQ To SQL 查询数据库中的数据

在 LINQ to SQL 中，关系数据库的数据模型映射到用开发人员所用的编程语言表示的对象模型。当应用程序运行时，LINQ to SQL 会将对象模型中的语言集成查询转换为 SQL，然后将它们发送到数据库进行执行。当数据库返回结果时，LINQ to SQL 会将它们转换回编程语言处理的对象。本节通过一个实例来介绍如何通过 LINQ to SQL 实现对数据库数据的查询。

注意：本实例使用的数据库是数据库文件，需要确保运行本实例的计算机安装了 SQL Server Express 程序。

(1) 创建一个控制台应用程序，命名为 LINQSqlDemo。

(2) 在“解决方案资源管理器”窗口中，右键单击 LINQSqlDemo 项目节点，单击“添加”|“新建项”菜单，在弹出的“添加新项—LINQSqlDemo”对话框中选择“已安装的模板”区域中选择“数据”选项，在右侧“模板”列表中选择“基于服务的数据库”模板，然后在“名称”文本框中输入 DemoDataBase.mdf，然后单击“添加”按钮，如图 30.5 所示。该步骤会添加一个 DemoDataBase.mdf 数据库文件，以此作为实例的查询数据库。

(3) 添加数据表以及数据。双击 DemoDataBase.mdf 文件，Visual Studio 2010 会弹出“服务器资源管理器”窗口。分别展开节点“数据连接”| DemoDataBase.mdf，右键单击“表”节点，在弹出的右键菜单中单击“添加新表”菜单，如图 30.6 所示。在弹出的创建数据表

窗口中，分别添加 ID、Name 和 Sex 等 3 个字段，类型分别为 int、varchar（50）和 bit 类型，其中 ID 列为主键。创建完毕的数据表定义如图 30.7 所示。



图 30.5 添加数据库文件



图 30.6 添加数据库新表

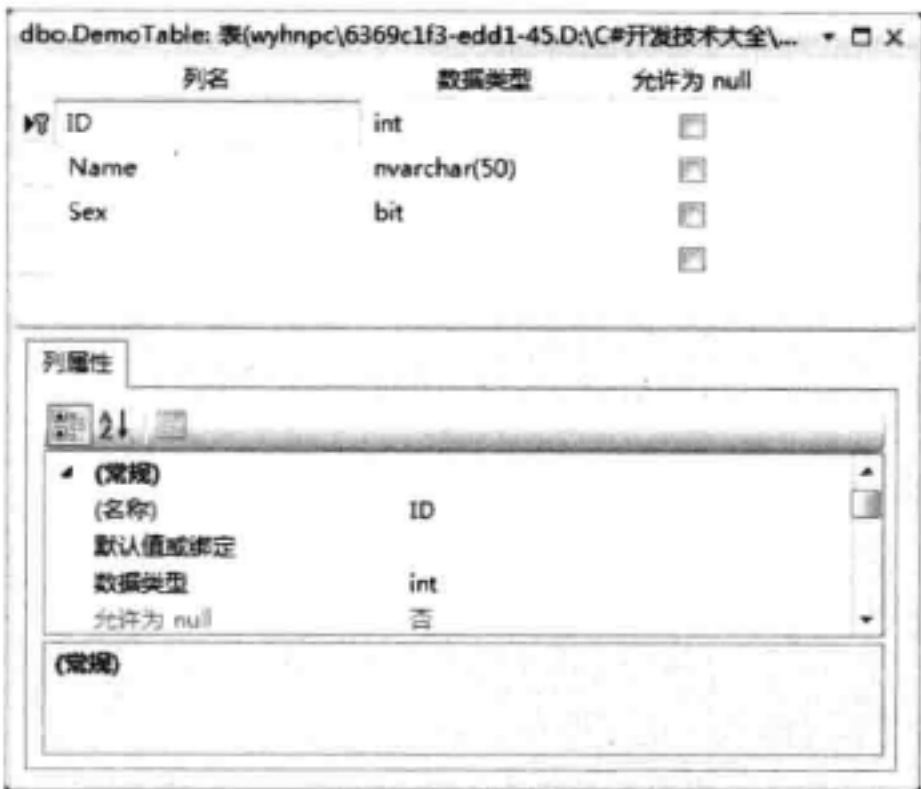


图 30.7 定义数据库新表

单击菜单栏的“保存”按钮，在弹出的“选择名称”对话框中，输入表的名称为 DemoTable。

(4) 在 DemoTable 中添加实例数据。右键单击 DemoTable 结点，单击“显示表数据”菜单。在弹出的表内容中，输入实例数据，数据内容如表 30.1 所示。

表 30.1 DemoTable表的实例数据

ID	Name	Sex
1	Tom	True
2	John	True
3	Jack	True
4	Mike	True
5	Mary	False
6	Kate	False
7	Jerry	False

(5) 添加数据库表的类定义。双击 Program.cs 文件，添加如下代码：

```
[Table(Name = "DemoTable")]
public class Person
{
    private int _ID;
    //定义属性 ID, 绑定数据库列名
    [Column(Name = "ID")]
    public int ID
    {
        //读取 ID 属性
        get
        {
            return this._ID;
        }
        //设置 ID 属性
        set
        {
            this._ID = value;
        }
    }

    private string _Name;
    //定义属性 Name, 绑定数据库列名
    [Column(Name = "Name")]
    public string Name
    {
        //读取 Name 属性
        get
        {
            return this._Name;
        }
        //设置 Name 属性
        set
        {
            this._Name = value;
        }
    }

    private bool _Sex;
    //定义属性 Sex, 绑定数据库列名
    [Column(Name = "Sex")]
    public bool Sex
    {
        //读取 Sex 属性
        get
        {
            return this._Sex;
        }
        //设置 Sex 属性
        set
        {
            this._Sex = value;
        }
    }
}
```

在代码中，定义了一个 Person 类，并使用了属性 [Table(Name = "DemoTable")] 来表明该类对应的数据库表名称。同样，在类中定义了 3 个私有字段和 3 个公有属性。每个公有

属性都通过[Column(Name = "××")]的语法进行标注,表明该属性对应的是数据表的某一行。


(6) 在 Main()函数中,添加以下用于查询的代码:

```
static void Main(string[] args)
{
    //定义数据源
    string path = System.Environment.CurrentDirectory;
    path = path.Substring(0, path.Length - 9);

    DataContext db = new DataContext(path + "\\DemoDataBase.mdf");
    //获取数据表中的数据
    Table<Person> persons = db.GetTable<Person>();
    //输出查询日志
    db.Log = Console.Out;
    //定义查询变量
    IQueryable<Person> custQuery =
        from person in persons           //指定查询对象
        where person.Sex == true         //指定查询条件
        select person;                   //选择查询结果

    Console.WriteLine("共查询{0}个结果",
        custQuery.Count().ToString()); //显示查询结果
    Console.WriteLine(custQuery.Count()); //显示查询数量
    //遍历查询结果
    foreach (Person cust in custQuery)
    {
        Console.WriteLine("ID={0}, Name={1}", cust.ID, cust.Name); ;
        //显示查询结果
    }

    Console.ReadLine();
}
```

 注意: 代码中的 DemoDataBase.mdf 文件的路径应随着文件所在目录的不同而有所区别,本实例中的路径为运行实例时 mdF 文件所在的目录。

代码中,首先通过 DataContext 类定义了一个用于数据库连接的实例,然后通过 GetTable()方法获取数据表对应的对象。通过 IQueryable 接口定义查询变量,然后通过 LINQ 查询语法构建查询表达式,最后通过 foreach 遍历查询的结果。

(7) 运行该实例,结果如图 30.8 所示。



图 30.8 LINQ To SQL 实例运行结果

30.3.2 用 LINQ To DataSet 查询缓存在 DataSet 中的数据

使用 LINQ to DataSet 可以更快、更容易查询在 DataSet 对象中缓存的数据,使开发人员能够使用编程语言本身而不是通过使用单独的查询语言来编写查询,因为 LINQ to DataSet 可以简化查询。LINQ to DataSet 也可用于查询从一个或多个数据源合并的数据。这可以使许多需要灵活表示和处理数据的方案能够实现。本节将通过一个实例来介绍如何实现 LINQ to DataSet 应用。

(1) 创建一个控制台应用程序,命名为 LINQToADODemo。

(2) 打开 Program.cs 文件,添加以下代码:

```
//定义数据源获取方法
private static DataSet getDataSet()
{
    //定义数据集数据源
    DataSet ret = new DataSet();
    DataTable dataTable = new DataTable();
    //添加数据列定义
    dataTable.Columns.Add(new DataColumn("ID",
        Type.GetType("System.Int32")));
    dataTable.Columns.Add(new DataColumn("Name",
        Type.GetType("System.String")));
    dataTable.Columns.Add(new DataColumn("Sex",
        Type.GetType("System.Boolean")));
    //添加数据集数据
    for(int n=0;n<10;n++)
    {
        DataRow newRow = dataTable.NewRow(); //定义新行
        newRow["ID"] = n; //添加 ID 值
        newRow["Name"] = "Name"+n.ToString(); //添加 Name 值
        if(n%2==0)
            newRow["Sex"] = true; //添加 Sex 值
        else
            newRow["Sex"] = false;
        dataTable.Rows.Add(newRow); //添加新行数据
    }
    //将数据集添加到数据表中
    ret.Tables.Add(dataTable);
    return ret; //返回数据集
}
```

getDataSet 方法用于在内存中构建一个数据集对象,然后在该数据集中添加一个数据表,同时在表中添加 10 行数据,用于后续查询操作。

(3) 添加查询代码。在 Main()方法中,添加以下代码:

```
static void Main(string[] args)
{
    //获取内存数据集
    DataSet ds = getDataSet();
    //获取内存数据表
    DataTable persons = ds.Tables[0];
    //定义查询变量
```



```

var query =
    from person in persons.AsEnumerable()//指定查询对象
    where person.Field<bool>("Sex") == true//指定查询条件
    select new//选择查询结果
    {
        ID = person.Field<int>("ID"),
        Name = person.Field<string>("Name"),
        Sex = person.Field<bool>("Sex")
    };
//遍历查询结果
foreach (var item in query)
{
    //显示查询结果
    Console.WriteLine("ID: {0} Name: {1} Sex: {2}",
        item.ID,
        item.Name,
        item.Sex);
}
Console.ReadLine();
}

```

在 where 查询条件子句中，对 Sex 列进行条件查询，同时通过 Select 语句，将查询结果创建到一个对象查询变量中，然后通过 foreach 语句，执行该 LINQ 查询。

(4) 运行该项目，实例的结果如图 30.9 所示。



图 30.9 LINQ To DataSet 实例运行结果

30.3.3 用 LINQ To XML 查询 XML 中的数据

在很多环境中，XML 已经是被广泛采用为格式化数据的方式。在 Web 上，在配置文件、Microsoft Office Word/Excel 文件以及数据库中，都可以看到 XML。LINQ To XML 经过了重新设计，是最新的 XML 编程方法。它提供文档对象模型的内存文档修改功能，支持 LINQ 查询表达式。

本节将通过一个实例介绍如何实现 XML 的 LINQ 查询。

(1) 创建一个控制台应用程序，命名为 LINQXMLDemo。

(2) 创建一个实例 XML 文件，将其保存到项目的文件夹中，命名为 DemoXml.xml。

该 XML 文件的内容如下代码所示。

```

<Root>
  <Data>
    <Type>A</Type>
    <Quantity>3</Quantity>
    <Price>84.50</Price>
  </Data>

```

```

<Data>
  <Type>B</Type>
  <Quantity>1</Quantity>
  <Price>69.99</Price>
</Data>
<Data>
  <Type>A</Type>
  <Quantity>5</Quantity>
  <Price>45.95</Price>
</Data>
<Data>
  <Type>A</Type>
  <Quantity>3</Quantity>
  <Price>88.00</Price>
</Data>
<Data>
  <Type>B</Type>
  <Quantity>10</Quantity>
  <Price>9.99</Price>
</Data>
<Data>
  <Type>A</Type>
  <Quantity>15</Quantity>
  <Price>24.00</Price>
</Data>
<Data>
  <Type>B</Type>
  <Quantity>8</Quantity>
  <Price>36.99</Price>
</Data>
</Root>

```

(3) 打开 Program.cs 文件，在 Main() 方法中，添加以下代码：

```

static void Main(string[] args)
{
    //加载 XML 数据
    XElement root = XElement.Load("../..\\DemoXml.xml");
    //定义数据查询变量
    IEnumerable<decimal> prices =
        from el in root.Elements("Data")           //指定查询对象
        let price = (decimal)el.Element("Price")   //指定查询条件
        orderby price                               //价格排序
        select price;                               //选择查询结果
    //遍历查询结果
    foreach (decimal el in prices)
    {
        Console.WriteLine(el);                     //显示查询结果
        Console.ReadLine();
    }
}

```

该段代码中，首先通过 XElement 加载了一个 XML 文件，然后通过 LINQ 查询语法，对 XML 文件中的 Price 元素进行排序，并将查询结果显示到控制台中。

(4) 运行该项目，实例的结果如图 30.10 所示。

30.3.4 用 LINQ To Objects 查询可枚举的集合

LINQ To Objects 是指直接对任意 IEnumerable 集合使用 LINQ 查询，无须使用中间

LINQ 提供程序或 API。可以使用 LINQ 来查询任何可枚举的集合，如 List、Array 或 Dictionary。该集合可以是用户定义的集合，也可以是 .NET Framework API 返回的集合。



图 30.10 LINQ To XML 实例运行结果

本节将通过一个实例来介绍如何实现 Object 的 LINQ 查询。

- (1) 创建一个控制台应用程序，命名为 LINQObjectDemo。
- (2) 打开 Program.cs 文件，在其中添加一个类定义，代码如下：

```
public class Student
{
    //定义 FirstName 属性
    public string FirstName { get; set; }
    //定义 LastName 属性
    public string LastName { get; set; }
    //定义 Scores 属性
    public int[] Scores { get; set; }
}
```

该类中，包含 3 个属性，其中的 Scores 属性是一个整型的集合。

- (3) 打开 Program.cs 文件，在 Main() 方法中，添加以下代码：

```
static void Main(string[] args)
{
    //定义数据源对象
    ArrayList arrList = new ArrayList();
    //添加数据源中的对象
    arrList.Add(
        new Student { FirstName = "FirstName1", LastName = "LastName1",
                      Scores = new int[] { 75, 84, }
        });
    //添加数据源中的对象
    arrList.Add(
        new Student { FirstName = "FirstName2", LastName = "LastName2",
                      Scores = new int[] { 97, 89 }
        });
    //添加数据源中的对象
    arrList.Add(
        new Student { FirstName = "FirstName3", LastName = "LastName3",
                      Scores = new int[] { 88, 94 }
        });
    //添加数据源中的对象
    arrList.Add(
        new Student { FirstName = "FirstName4", LastName = "LastName4",
                      Scores = new int[] { 98, 92 }
        });
    //定义查询变量
```

```

var query = from Student student in arrayList //指定查询对象
            where student.Scores[0] > 95      //指定查询条件
            select student;                  //选择查询结果
//遍历查询结果
foreach (Student s in query)
    Console.WriteLine(s.FirstName+" "+s.LastName + ": " + s.
        Scores[0]);                          //选择查询结果

Console.ReadKey();

}

```

在该段代码中，首先定义了一个 ArrayList 对象，作为查询的数据源。在该对象数据源中，添加 4 个 Student 类的实例。然后使用 LINQ 查询语法，查询 Scores 集合中第一个元素大于 95 的所有记录，然后将该集合中的实例信息，显示到控制台中。

(4) 运行该项目，结果如图 30.11 所示。

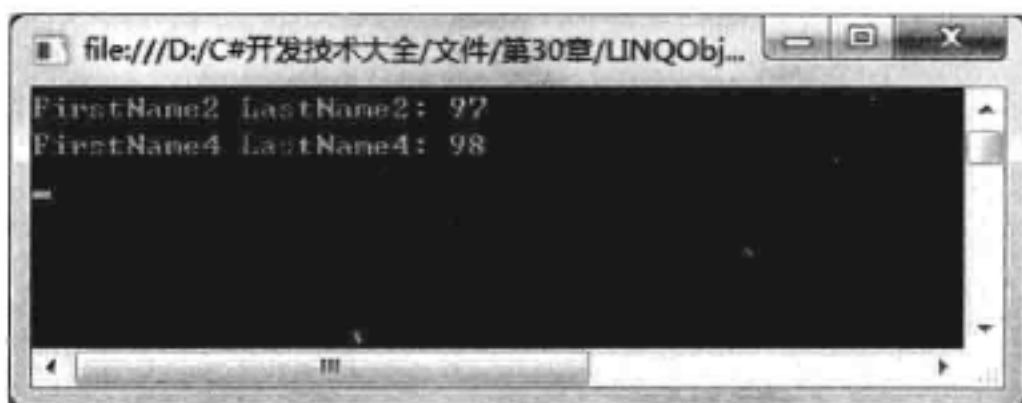


图 30.11 LINQ To Object 实例运行结果

30.4 本章总结

语言集成查询 LINQ 改变了以往对于对象的数据查询的操作方式，通过该框架提供的查询功能，开发人员可以高效率地实现各种查询功能。本章通过对 LINQ 的概述、语法、模型查询等内容，详细地介绍了 LINQ 的概念和使用，使读者可以更加方便地使用 .NET 框架中的查询模型。

30.5 实战练习

1. 在 Visual Studio 2010 中新建一个控制台应用程序，在程序中创建一个字符串数组，包含多个字符串，然后使用 LINQ 查询找出以字母 T 开头的字符串，并输出到控制台。
2. 在 Visual Studio 2010 中新建一个控制台应用程序，在该项目中创建一个保存学生成绩的数据库，往数据库中输入测试信息，使用 LINQ 查询显示学生成绩。
3. 接第 2 题，继续编写代码，用 LINQ 从学生成绩表中查询成绩超过 90 分的学生，再用 LINQ 查询成绩低于 60 分的学生，然后将这两类学生的成绩合并输出。